Intermediate Data Structures and Algorithms

Mark Eramian

Course readings for CMPT 280

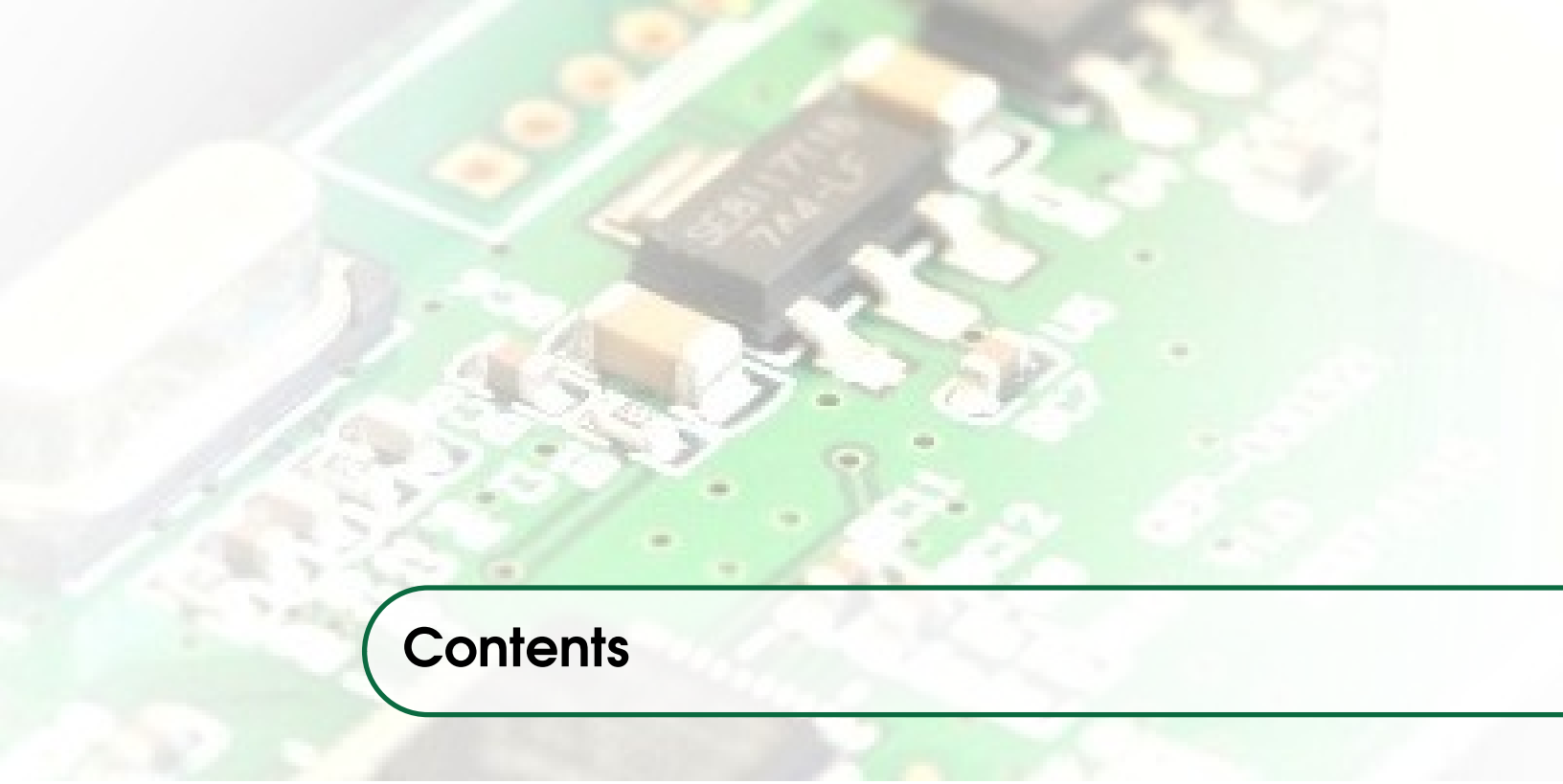
PRODUCED BY THE AUTHOR FOR STUDENTS ENROLLED IN CMPT 280 AT THE UNIVERSITY OF SASKATCHEWAN.

Second edition, January 2015

Acknowledgements

My appreciation and thanks go to these students:

- Mr. G. Scott Johnson for his help with locating small errors in typography and content;
- Mr. David Kreiser for spotting many additional typos; and
- Ms. Janelle Hindman for pointing out some subtle typos.



Contents

1 — Lists

Learning Objectives

After studying this chapter, a student should be able to:

- describe what a list is;
- identify two different ways a list might be implemented;
- explain how new items are added and removed at the beginning of an array-based list; and
- explain how new items are added and removed at the beginning of a linked list.

1.1 Abstract Data Type: List

A list is an abstract data type[1] (ADT) that manages a *collection* of data with a linear ordering. The ordering may or may not be meaningful in terms of the data elements themselves. That is to say, each element in the list has a *position* (e.g. first element, third element, etc.) but that position may or may not have any meaning with respect to the data in the elements themselves. If the list is sorted, for example, then the position of an element has meaning; the fifth element in the list is the fifth smallest element. Otherwise, the position of an element indicates only the element's rank in the list order.

At a minimum, a list typically supports the following operations:

- Create a new, empty list.
- Insert a new element at the front of the list.
- Delete the first item in the list.
- Obtain the first item in the list.

We could easily imagine more operations (and later, we will!), but these provide the bare minimum.

Note: listing the desired operations of an data structure, as above, a pretty bad way of specifying a data structure. For example we haven't said anything about what should happen if you try to obtain

[1] Abstract data types (ADTs) were covered in CMPT 115, so we will save the review of the formal definition for later. For now we simply remind you that an ADT is a specification of data and operations on that data that is free of implementation details.

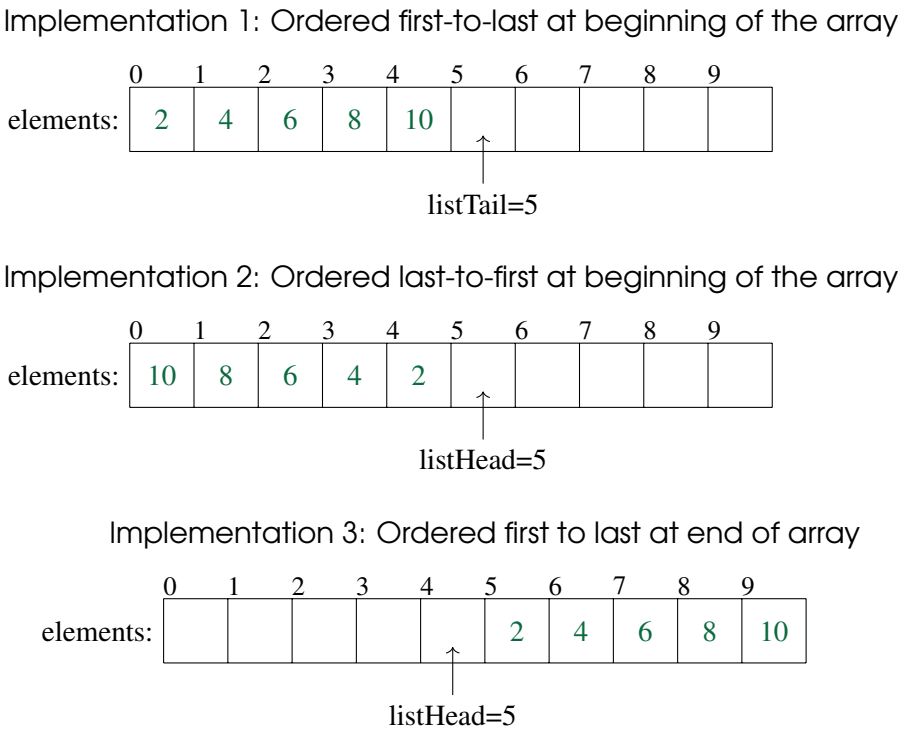


Figure 1.1: Three ways of implementing an array-based list. The array `elements` stores the items in the list. The variable `listHead` or `listTail` are integer variables that contain the offset of the array immediately before/after the head/tail of the list depending on the situation.

the first item in a list that is empty. We will return to the question of specifying data structures without ambiguity later in the course. The above list of operations will do for now.

1.2 List Implementations

There are two main approaches for implementing a list: an array, or a linked structure (a.k.a. a *linked list*).

1.2.1 Array-based Lists

Arrays seem like a natural choice for lists. They implicitly provide the required linear ordering — you just place each element at consecutive offsets in the array. There are actually a few choices of how to do this. Three ways of storing the list of integers $\{2,4,6,8,10\}$ in an array are shown in Figure 1.1. The different choices can have great effect on the efficiency of the operations on the data structure as we shall see momentarily.

Adding an Element at the Beginning of an Array-based List.

Consider the operation that inserts an element at the beginning of the list. In Implementation 1 at the top of Figure 1.1, which is the natural choice that we would probably all make if we didn't really think about it, we can see that if a new element were to be inserted at the beginning of the

list, we would have to move all of the existing elements over by one array offset to make room. In Implementation 2, the list is always read from right to left, so the first element in the list is actually the element '2,' and when a new element is added at the beginning of the list, it should become the new first element of the list. A new element would be inserted to the right of the '2,' at offset 5, which would not require any other elements to move. Implementation 3 is similar, where any new element simply gets added to the left of the left-most element in the array, and no other elements must move. Thus, it is easy to see that for a large list, this choice could have significant impact on performance.

Suppose we want to program implementation 3. We would need an array (called `elements` in Figure 1.1), and an integer variable that always contains the offset of the array that is immediately before the first element of the list (called `listHead` in Figure 1.1). The algorithm for inserting a new element at the beginning of the list would be:

```
Algorithm insertFirst( newElement )  % implementation 3
newElement is the element to add to the beginning of the list.

if(listHead < 0)
    list is full, throw exception
else
    elements[listHead] = newElement;
    listHead = listHead - 1;
end
```

Essentially, we first make sure there is a space in the array, if there is, we put the element in the empty space at the start of the list (always indexed by `listHead`) and move `listHead` to the next open space. Note that this operation doesn't require any shifting of existing elements in the array. For Implementation 1, however, we would have to shift all n of the existing elements over by 1 to make room for the new first element, which is much slower. If you remember your Big-O notation, our pseudocode for `insertFirst` for Implementation 3 is $O(1)$, whereas implementation 1 would be $O(n)$. If you don't remember your Big-O notation, don't worry, we'll review it soon, but it should still be clear why Implementation 1 is inferior.

1.2.2 Linked Lists

Linked lists are a data structure that you would have seen in your CMPT 115 (or equivalent) course, but here is a quick review.

A linked list consists of a series of *nodes* in a linear order. Each node represents a different position in the list. Each node contains two pieces of data: the data element at the position in the list represented by the node, and a reference to the node representing the next position in the list. The node representing the last position in the list contains a "null" reference for the next node, indicating that there is no next node. In an object-oriented implementation, the nodes would instances of one type of object. The complete list itself would be represented by a different object. This list object stores a reference to the first list node and possibly other information, such as the number of elements in the list (see Figure 1.2) .

Adding a New Node at the Beginning of a Linked List

Inserting a node at the beginning of the list requires that some references be updated. The reference to the first node of the list in the List object must be updated to point to the new node. The "next node" reference in the new node must be updated to point to the old first node. This process is illustrated in Figure 1.3.

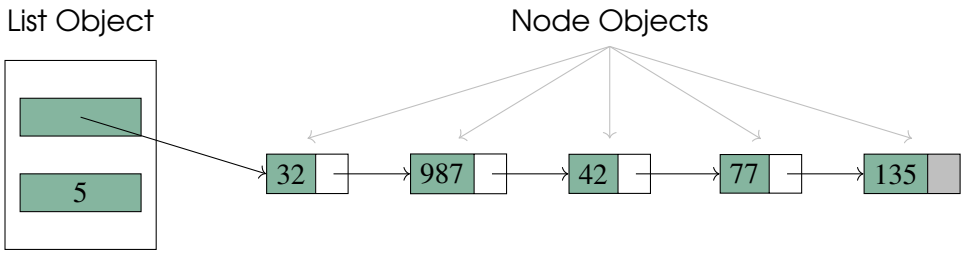


Figure 1.2: An object-oriented simple linked list containing five data elements. The list object contains a reference to the first list node, and the number of nodes in the list. The node objects contain the data element and a reference to the next node. The last node's reference is shaded grey to indicate that it is "null" and has no successor.

Removing a New Node at the Beginning of a Linked List

Removing the element at the beginning of the list also requires that two references be updated. First, the reference in the List object to the first node in the list has to be updated to point to the second node. Second, the reference in the first node has to be updated to null, to "unlink" it from the list. This process is illustrated in Figure 1.4.

1.3 Linked List Variants

There are many variations on this basic idea of linked lists. One you may have heard of is the doubly linked list. In a doubly linked list, each node has a reference not only to its successor, but also to its predecessor. This facilitates more efficient insertion and deletion from the end of a linked list.

Linked lists may also be *circular*. A circular linked list is one where the last node in the list, rather than containing a "null" reference, refers instead back to the first node in the list. Circular lists can be singly linked or doubly linked (in which case the first node references the last node).

While we won't review the details of these lists here, we may look at some of them in the course of doing other things.

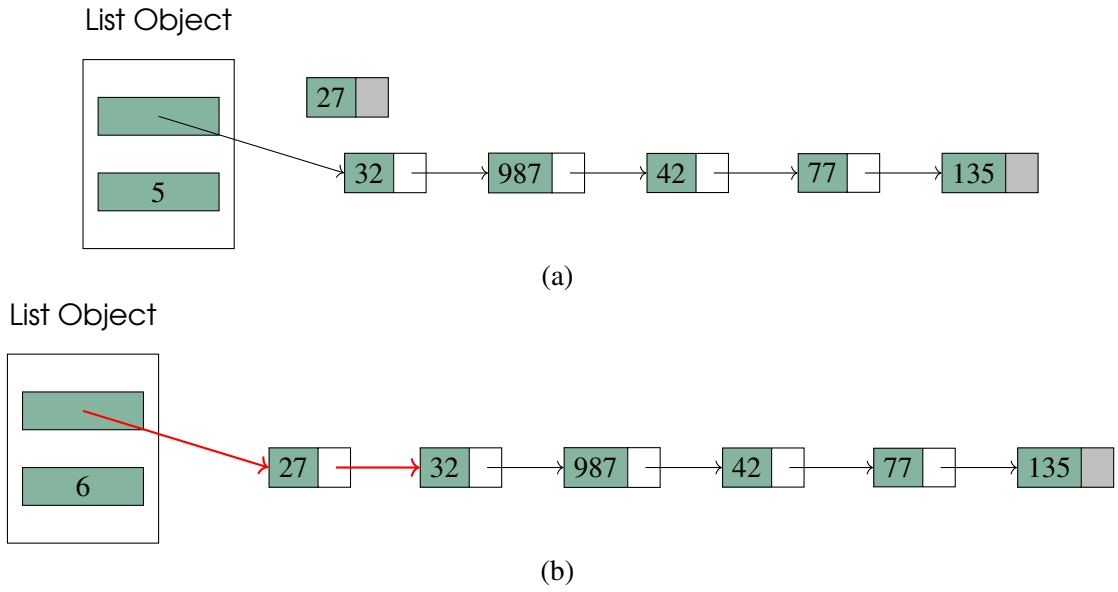


Figure 1.3: Inserting an new node containing the element 27 at the beginning of the list; a) before references are adjusted, b) after references are adjusted. Red arrows show references that were updated.

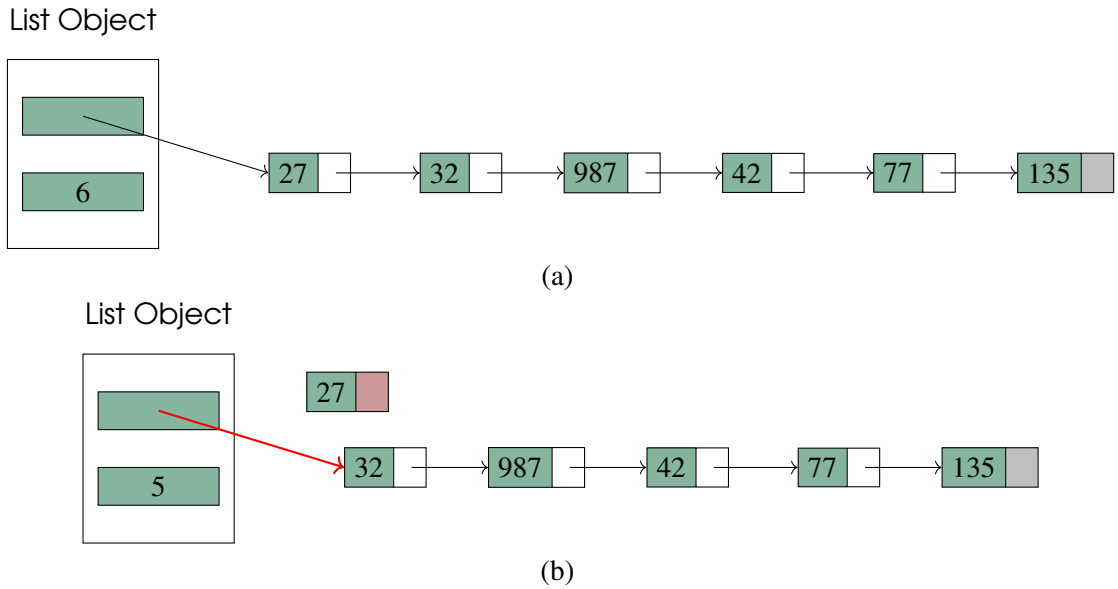
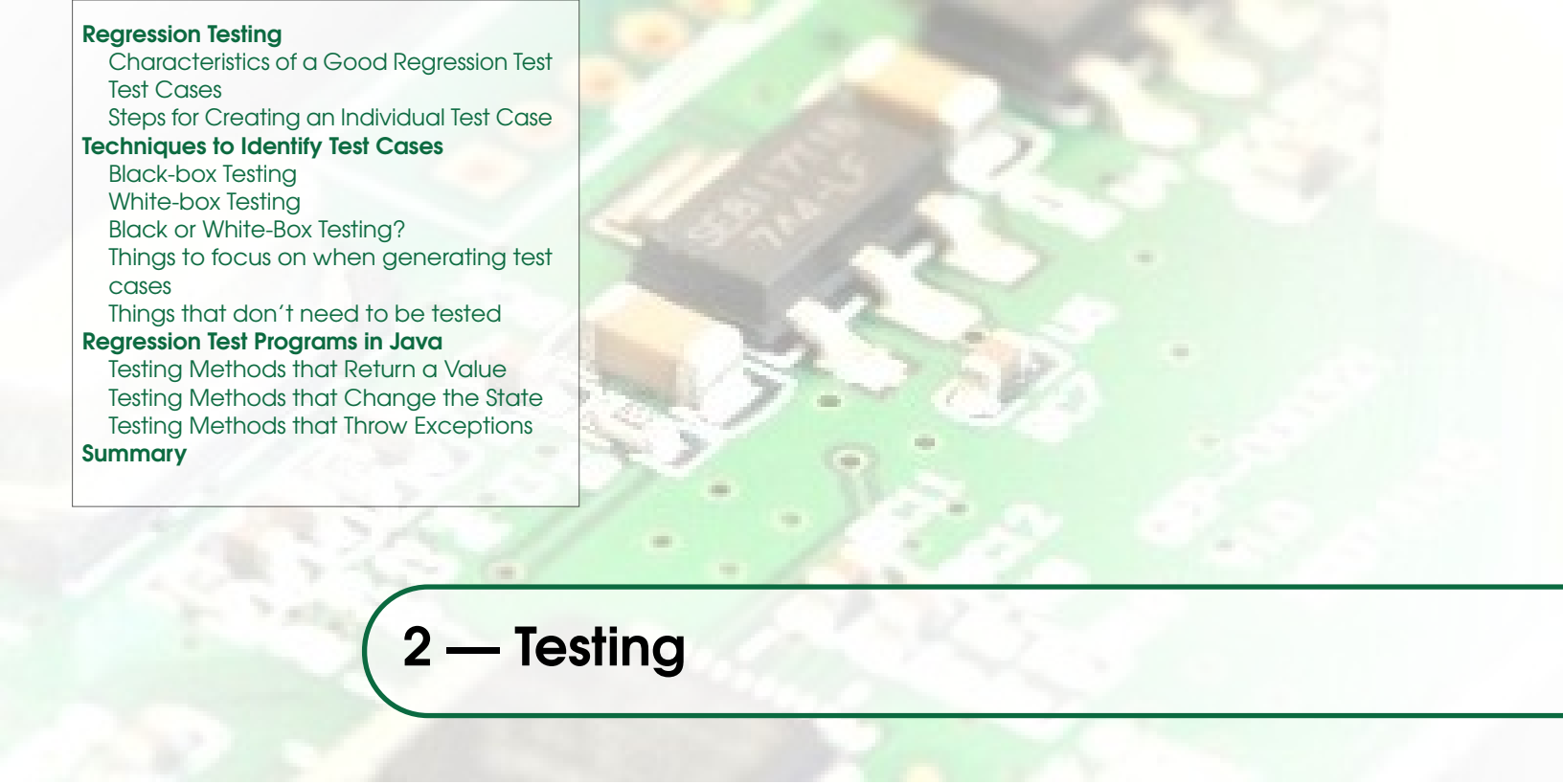


Figure 1.4: Removing the first element of a list; a) before references are adjusted, b) after references are adjusted. Red arrows and shading show references that were updated.



2 — Testing

Learning Objectives

After studying this chapter, a student should be able to:

- explain the purpose and goals of regression testing;
- describe the characteristics of a good regression test program;
- describe the white-box and black-box test case generation strategies;
- generate test cases using both the white-box and black-box approaches;
- write code for test cases that require checking a return value;
- write code for test cases that require verifying a change of state; and
- write code for test cases that may throw exceptions, expected or unexpected.

2.1 Regression Testing

Good testing habits will be essential in this course. We will be building data structures out of other data structures, so we must be certain that we develop good unit tests for all of our objects, so that we can use them to create new objects with confidence that the existing objects are problem-free.

Regression testing is a strategy which allows tests to be quickly re-run on modules of a software project that change. Each object (Java class) that we write in this course (at least on assignments) will include a *regression test*.

A regression test is an automated test program that is designed to uncover new bugs (i.e. regressions) after code has been changed to make enhancements, fix bugs, etc. In an object oriented language, each object can have a regression test which verifies the correct operation of each of the object's methods. One could also imagine regression tests written for groups of objects that work together, at higher and higher levels of abstraction. In this course, since we will be mostly programming data structures, we will focus on regression testing at the level of individual objects.

2.1.1 Characteristics of a Good Regression Test

A good regression tests has the following characteristics:

Thorough testing: All methods are tested thoroughly to ensure that they work in all situations.

Easy to run: Good regression tests are easily re-run when a change is made to an object. The input data for the tests should either hard-coded or read from a file. The regression test code should handle all setup for each test case. Expected results should also be hard-coded or read from a file, and should be automatically compared with the actual results by the test program. The regression test program need not report successful tests, and should produce output only when a problem is detected so that one doesn't have to hunt through a large number of successful test reports for that one unsuccessful test. Exceptions to this rule are to report progress (e.g. percent complete) if the tests take a long time to run, or to print out the output of a test that can be easily checked for correctness at a glance by a human.

Easily expanded: Good regression test programs should be easily expanded to include new test cases when additional test cases are discovered, or tests for newly added functionality need to be incorporated into the test program.

2.1.2 Test Cases

A regression test is composed of a number of individual test cases. A test case tests a particular aspect of a method of the object. For example, suppose a linked list class has a method called `isEmpty()` which returns `true` if the list is empty, and `false` otherwise. This extremely simple function gives rise to at least two test cases. We must ensure that the method returns `true` when the list is, in fact, empty, and returns `false` when the list has one or more items in it. To be extremely thorough, we might want to create test cases for `isEmpty()` for when the list has 0, 1, 2, and "many" items in the list. For each test case, we would arrange for the list to have the right number of items, call `isEmpty()`, and check if it returned the expected result.

In this class, most of the time you will be generating test cases to test particular aspects of the methods of data structure objects.

2.1.3 Steps for Creating an Individual Test Case

Follow these steps to generate a test case.

1. Identify the test case or situation (e.g. does `isEmpty()` return the right value when the list is empty).
2. Determine how to set up the test case. Hard-code the necessary input data, and set up the program's state as needed for the test case. For example, if we want to test whether our linked list's `isEmpty()` method operates correctly when there are two items in the linked list, we would note that we would have to create two elements to insert into the list (i.e. hard-code the input data), and then insert them into the list (i.e. set up the program state for the test) to create the proper conditions for the test.
3. Determine the expected result. Test cases should be simple enough that the expected result can be determined manually. Continuing the example from steps 2, we would take note that when a linked list contains two elements, the `isEmpty()` method should return `false`.
4. Code the test. Write the actual program that sets up the program state to that of the test conditions. Again, continuing our running example, we would write code that declares and initialize two elements of the type that our linked list may contain, and insert them into a list. Then we would write code to call `isEmpty()` on that list, test whether its actual return value was the expected return value of `false`, and to output a message indicating a failed test case if the result was not the expected one.

5. Execute the test program. Run the test program to make sure that the test does not fail.

Note that when we code tests, we assume that all methods that are called by the method being tested are *already* tested and verified to be correct. The only exception is when the method being tested is recursive.

2.2 Techniques to Identify Test Cases

Two common strategies for identifying test cases are black-box and white-box testing.

2.2.1 Black-box Testing

For black-box testing, we identify expected inputs and outputs without looking at the code for the method we are writing the regression test for. We can think of it as the code being inside a black (i.e. non-transparent) box, and we can't see inside. Instead, we base our tests on the public interface of the object or function, that is, what each method is supposed to do, and its expected inputs and outputs.

By way of example, suppose we were asked to write a regression test program for the `substring` method of Java's `String` object. First we would look at the public interface (click here!) for that method. We can see that this method takes two integer parameters, which are two integer indexes (offsets) of characters within the string, and returns the substring that starts at `beginIndex` and ends at `endIndex-1`. With this information we can use the black-box strategy to generate test cases. We always like to test boundary cases. So we would probably like to have a test case where `beginIndex` is 0, another where it is 1, another where it is the offset of a character in the middle of the string, another where it is the offset of the second-last character of the string, and yet another where it is the offset of the last character of the string. We'd probably also want similar test cases for `endIndex`. We would also want to test that the listed exceptions are correctly thrown when we provide erroneous input data, so that would mean having test cases where the indices are negative, and where they are greater than the length of the string. This sounds like a lot of test cases, and we could probably even think of more. The good news is that we can often cover more than one situation with a single test case. For example, we could create a `String` object containing the string `"Zombie"`, and call `substring` with a beginning index of 0 and an ending index of 6. This would cover the case where `beginIndex` is the offset of first character *and* the case where the `endIndex` is the offset of the last character.

2.2.2 White-box Testing

In white-box testing, we generate test cases directly from the code being tested. Think of the code being inside of a white (transparent) box that we can see inside of. White-box testing focuses on finding test cases that, together, execute all possible paths through the code being tested. Consider the following code which counts the number of times the string `s` occurs within the array of strings `a`:

```
public static int count(String[ ] a, String s)
{
        int count = 0;
        int i = 0;
        while (i < a.length)
        {
                if (s.equals(a[i]))
                        count = count + 1;
                i = i + 1;
        }
        return count;
}
```

We wish to consider all of the different paths through the program. So we must generate test cases that cause the `while` loop to execute 0 times, 1 time, 2 times, and "many" times. We would also want test cases where the condition in the `if` statement never evaluates to `true`, is only true once, is only true twice, and is always true. Note that the different paths through the program correspond to the different places in the array where the string `s` might occur, and we get much the same tests as we would have had we used black box testing to test when 's' is at index 0 in `a`, at index 1 in `a`, at index `a.length-1` in `a`, etc.

2.2.3 Black or White-Box Testing?

So which approach should we use and when? Many test cases will be discovered by both black-box and white-box testing, but some won't. So ideally, one would need to use both approaches all of the time.

2.2.4 Things to focus on when generating test cases

Focus on boundary cases, especially when size of the input matters. Test as many of size 0, 1, "many," "almost as big as possible," and "as big as possible" that apply. For loops, use test cases that cause the loop to run 0, 1, "many," "one less than maximum," and "maximum" number of times. For containers (like a list), test methods for their behaviour when the container is empty, and when the container is full. Test for correct behaviour when a method's preconditions are violated — in other words, make sure the right exceptions are thrown (or other error indications issued) when the method is given invalid input.

2.2.5 Things that don't need to be tested

It is not necessary to write test cases that don't compile, for example, the case where a method is called with an argument of the wrong type. The compiler is there, as your friend and ally, to catch these types of syntactical errors. Regression testing is concerned with discovering things that can go wrong at run-time.

2.3 Regression Test Programs in Java

In this course, we will mainly be writing regression tests for individual Java classes. Such a regression test should thoroughly test every method defined by the class. A regression test for a Java class can

be placed inside a `main()` function in the same `.java` file in which the class is defined.[1]

```
class Foo {
    // Instance variables
    ...

    // Method definitions
    ...

    // Regression test
    public static void main(String args[]) {
        // Code for test case implementations
        ...

    }
}
```

For each test case generated, the implementation of the test case (hard code the input, set up program state, run the test, check for the expected result) can be placed in this `main()` function. In the next sections we will review some typical patterns that arise when implementing test cases.

2.3.1 Testing Methods that Return a Value

For methods that return a value, one sets up the desired program state, calls the method, and checks whether the value returned is the value expected from the current program state. We illustrate this in Example 2.1.

■ **Example 2.1** In the following regression test, we are testing a method of the `ArrayedList` class. This class implements an array-based list (a list where elements are stored in an array) which we talked about in Chapter 1. First, a list that can contain a maximum of 5 elements is instantiated, and then the `isEmpty()` method is called to see whether it correctly returns `true`. If it doesn't, an error is printed to the console. Note that if the test of `isEmpty()` **does** return the expected result, no message is produced. Successful tests should succeed silently.

```
// Regression test
public static void main(String args[]) {
    // List with at most 5 elements
    ArrayedList<Double> L = new ArrayedList<Double>(5);

    // Test isEmpty() when the list is empty
    if( L.isEmpty() == false )
        System.out.println("ERROR: isEmpty() returned false on an empty list.");
}
```

■

2.3.2 Testing Methods that Change the State

Some methods don't return anything, but rather change the state of the program or, in our case, the data structure. In Example 2.2 we show two ways of checking whether a method that changes the state of the data structure has done so correctly. The first involves querying only the part of the state that changed, the second is to query the entire data structure.

[1] We could use a unit testing framework like JUnit, but for this course, it's overkill. We want to focus on generating good test cases, and not worry about all of the overhead of using JUnit.

■ **Example 2.2** In this example we are again using our array-based list that can hold up to 5 elements. This time, we are using the `insertFirst()` method to add an item to the beginning of the list. This changes the state of the list such that the newly inserted element, 2.5, should be the first item in the list. So in order to test whether `insertFirst()` worked correctly, we use the `firstItem()` method (which, in a complete test program, would have already been thoroughly tested!) to obtain the first item in the list and check whether it matches the element we just inserted. If `firstItem()` does not return 2.5, then something went wrong with `insertFirst()` so we report a failure. Here we have verified only the part of the state of the program that should have changed. This makes for an easy-to-write test case, but has the potential to miss other side effects. For example, we couldn't tell from this test whether any pre-existing elements in the list (if there were any) were erroneously affected or altered.

Then, the state is changed again, by calling `insertFirst()` a second time which adds a new element, 3.5, to the beginning of the list. At this point, we could check whether the first element of the list is now the newly inserted element 3.5. Instead we demonstrate a technique where we query the entire data structure and compare it visually to the expected result by printing the entire list, and the entire expected list one after the other. This approach can be very useful as long as the program state is small enough that it can be inspected and verified by a human very quickly.

```
// Regression test
public static void main(String args[]) throws Exception {
    // List with at most 5 elements
    ArrayedList<Double> L = new ArrayedList<Double>(5);

    // Test insertion on empty list (changes the state)
    L.insertFirst(2.5);
    Double x = L.firstItem();

    // Verify the state by querying part of the state.
    if( x != 2.5 )
        System.out.println("ERROR: expected firstItem() to be 2.5, got: "
                           + x);

    // Test insertion on list with one element (changes the state)
    L.insertFirst(3.5);

    // Verify the state by querying the whole data structure

    // Next line assumes L implements toString() which returns a string
    // representation of an object.
    System.out.println("Current list:  " + L);
    System.out.println("Expected List: 3.5, 2.5,");
}
```

■

2.3.3 Testing Methods that Throw Exceptions

For either of the previous two types of test case, we could have the additional complication that the method being tested may throw an exception in some circumstances. There are a number of possible cases:

1. An exception is expected, and it occurs.
2. An exception is expected, and no exception occurs.

3. An exception is expected, and a **different** exception occurs.
4. An exception is not expected, but an exception occurs.
5. An exception is not expected, and none occurs.

Cases 2, 3, and 4 would require that a testing failure message be printed. Cases 1 and 5 are "successful" tests, assuming that the program/data state after the test is also correct.

Cases 1-3: Test cases where an exception is expected.

Any test case that may throw an exception should be placed within a try-catch block. The test is run in the `try` block. If no exception occurs, an error is reported (this is case 1, above). If the expected exception occurs, we do nothing (this is case 2, above). If a **different** exception from the expected exception is caught, then we report an error (this is case 3, above). This is illustrated in the following code where we try to obtain the first item from a list that is empty.

```
public static void main(String args[]) {
    ArrayedList<Double> L = new ArrayedList<Double>(5);

    // Test firstItem() on empty list
    // (ContainerEmpty280Exception exception expected!)
    Double x;
    try {
        // Try to get the first item in the list (which happens to be empty)
        x = L.firstItem();
        // Case 1: an exception did not occur.
        System.out.println("ERROR: Exception expected but did not occur.");
    }
    catch(ContainerEmptyException e) {
        // Case 2: Expected exception, do nothing.
    }
    catch(Exception e) {
        // Case 3: Some other, unexpected exception occurred.
        System.out.println("ERROR: ContainerEmpty280Exception expected, " +
                   "a different exception occurred.");
    }
}
```

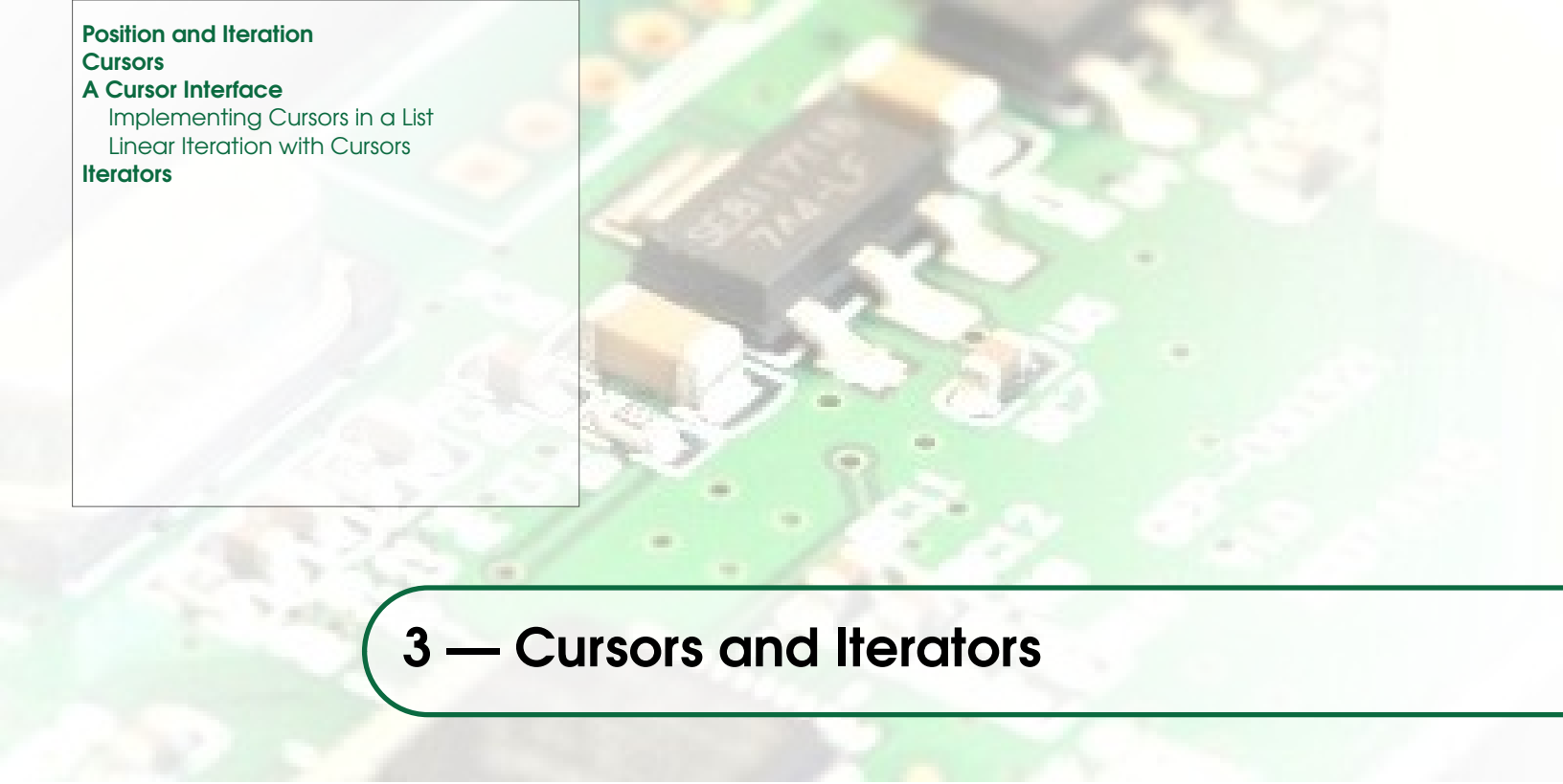
Cases 4 and 5: Test cases where an exception not expected.

If you are testing a method which is capable of throwing an exception, even if one is not expected, the test should handle the situation where an exception is erroneously produced. This is handled by putting the test inside of a try-catch block, and report an error if **any** exception is caught. Here is an example where we are testing the `insertFirst()` method on an empty list. No exception should occur, so if one occurs, we report an error. Then we use the `firstItem()` method to check whether the previously inserted item is now the first item in the list. No exception should occur, because the list should not be empty (we previously inserted an item), so if an exception does occur as a result of test, the test has failed, and an error must be reported. Even if an exception does not occur though, we still must verify the return value of `firstItem()` to make sure it is correct. If it isn't, then the test fails.

```
public static void main(String args[])  {
    ArrayedList<Double> L = new ArrayedList<Double>(5);

    // Test firstItem() on non-empty list (exception not expected!)
    try {
        // Case 5: no exception, as expected.
        L.insertFirst(2.5);
    }
    catch( Exception e ) {
        // Case 4: Unexpected exception.
        System.out.println("ERROR: insertFirst() on an empty list " +
                           " threw an exception.");
    }
    Double x;
    try {
        x = L.firstItem();    // List is not empty, no exception expected
        // Case 5: No exception, as expected.
        // But the test could still fail based on the state of the list
        // if the element returned is not 2.5...
        if( x != 2.5 )
            System.out.println("ERROR: expected firstItem() to be 2.5, got "
                               + x);
    }
    catch( Exception e ) {
        // Case 4: Unexpected exception.
        System.out.println("ERROR: firstItem() on a non-empty list " +
                           " threw an exception.");
    }
}
```

2.4 Summary

Writing thorough regression tests is good. It will save you a lot of anguish by allowing you to be confident that an object's implementation is correct before you use it in another object.



3 — Cursors and Iterators

Learning Objectives

After studying this chapter, a student should be able to:

- explain what a cursor is;
- explain the valid cursor positions within a linear data structure;
- explain how to represent a cursor's position within a linked list data structure;
- list the operations that can be performed on data structures with cursors;
- demonstrate how to iterate over the elements in a data collection using the cursor interface;
- explain the similarities and differences between a cursor and an iterator object.

3.1 Position and Iteration

A *collection* is a data structure that is comprised of a set of data elements in some conceptual arrangement (e.g. in a linear order in the case of a list). A list is a collection because it is a set of data elements stored with a total linear ordering. A great many data structures are collections, and most of the data structures we study in this course are collections.

It is very common to want to look at each element in a collection and do something to it or with it. This is achieved using *iteration*: we "loop" over all of the elements in a collection and process each one in turn. In this chapter, we look at how we can abstract this process and construct a common interface for iteration over the elements in a collection regardless of the underlying arrangement. This will be very useful when we consider collections of data where there is no explicit linear ordering (e.g. a tree). It is therefore useful to have the notion of a *position* within data structure.

3.2 Cursors

We will represent a position within a collection with an abstraction called a *cursor*. A *cursor* is a property of a collection that records a specific position. Since a cursor is a property of a collection object, it is implemented using instance variables and manipulated using instance methods. A

particular collection might have any number of cursors, but in our initial examples we will consider data structures with only one cursor.

A cursor records a single position. A cursor may be positioned at an element, or it may not be positioned at any element. Let's consider the list of integers below (it doesn't matter if it's array-based or linked, the discussion is the same either way):

1 2 3 4 5

A cursor may be positioned at one of the data elements, e.g. element 2 (the green underline denotes the cursor):

1 2 3 4 5

A cursor could be on the *first* item:

1 2 3 4 5

A cursor could be on the *last* item:

1 2 3 4 5

Or a cursor might be positioned *before* any of the elements:

1 2 3 4 5

Finally, a cursor might be positioned *after* any of the elements:

1 2 3 4 5

3.3 A Cursor Interface

Here are the operations that we might want to perform on a cursor. These operations could form a cursor interface.

itemExists	Determine whether or not the cursor is positioned at an element, and therefore that there exists an item at the cursor's position.
item	Return the element in the container at which the cursor is positioned.
goFirst	Move the cursor to the first element.
goForth	Move the cursor to the next element.
goLast	Move the cursor to the last element.
goBefore	Move the cursor to the position before the first element.
goAfter	Move the cursor to the position after the last element.
before	Test whether the cursor is positioned before the first element.
after	Test whether the cursor is positioned after the last element.

Note that a cursor need not provide **all** of these operations. Just because an ADT has a cursor doesn't necessarily mean that it should provide public methods to manipulate the cursor. A good example of this is a *stack* ADT. The "current position" in a stack is always the top of the stack and stack ADTs generally only provide the user with access to the element at the top of the stack. We wouldn't want the user to manipulate this cursor and give themselves access to other elements not at the top of the stack. So a stack ADT would likely only provide the methods `item()` and `itemExists()` in its interface.

3.3.1 Implementing Cursors in a List

In this section we'll get a taste of how one might implement a cursor in a linked list. For a list, these are the valid cursor positions: at one of the elements, before the first element, and after the last element. The special cursor positions "before" and "after" are often the trickiest to deal with because it is not always obvious how to represent them; it will be different for each data structure.

It's fairly easy to represent the cursor position when it is positioned at an element; we can just store a reference to the node at which the cursor is positioned. So to store the cursor position, an instance variable of type `LinkedNode<I>` will do. But then, how do we represent the "before" and "after" positions? The most natural thought is to interpret `null` as the "before" or "after" position. But wait... we can't use `null` to represent both the before **and** after positions. The solution is to use two variables to represent the cursor position, one called `position` that records the current cursor position, and one called `prevPosition` that records the cursor position immediately prior to the current one. Then, all positions can be represented:

`position` value	`prevPosition` value	# elements in list	Actual position
non-null	non-null	> 1	between 2nd and last element
non-null	null	> 0	"first"
null	non-null	> 0	"after"
null	null	> 0	"before"
null	null	0	"before" **and** "after"

Here's how we might implement the `goFirst()` method for the `LinkedList` class:

```
class LinkedList<I> {
    ...

    // These variables represent the cursor position.
    LinkedNode<I> position;
    LinkedNode<I> prevPosition;

    // And here's how we would implement goFirst()...
    public void goFirst() throws Exception {

        // goFirst() only makes sense if the list is non-empty...
        if( !this.isEmpty() ) {

            // Set the cursor position to the first list element
            this.position = this.head;

            // Set the position preceding the cursor position
            // to null since no element comes before the first one.
            this.prevPosition = null;
        }
        else throw new Exception("Cannot position cursor at the first " +
                                 "element of an empty list.");
    }

    ...
}
```

So now, give some thought to how to represent a cursor position in an array-based list. You might need to do this at some point (hint, hint!).

3.3.2 Linear Iteration with Cursors

Let's suppose that both our `ArrayedList` and `LinkedList` classes (discussed at length in class) implement this cursor interface. Let's further suppose that we have a list of numbers and we want to compute their sum. We could store the numbers in either type of list and write code that uses the cursor interface to "loop" over all of the elements and add them up. But regardless of which type of list we choose, **this code will be the same** despite the fact that the two lists store the elements in completely different ways. This is because the cursor interface is the same for both types of list. We illustrate with the following code fragment:

```
// Suppose L is a list of Double, initialized to contain
// zero or more elements.

Double sum = 0;

L.goFirst();
while(L.itemExists()) {
    sum = sum + L.item();
    L.goForth();
}
```

This code will work regardless of whether `L` is an array-based or linked list. This is the great advantage of cursors. It makes iteration over a collection work the same way regardless of the internals of the data structure. In order for this to happen, the internal implementations of the cursor operations themselves will be different for each type of collection, but they provide the same outward functionality. This is a perfect example of encapsulation.

3.4 Iterators

An *iterator* is very similar to a cursor with one important difference. Like a cursor, an iterator records a position within a collection. But, while a cursor is a property of a collection, an iterator is a *distinct object*. It is a separate object that represents a position in an instance of a collection. Another major difference between iterators and cursors is that iterators **always** provide public methods for iterating over all of the elements in the collection. A cursor, by definition, is just an abstraction of "position." An iterator, by definition, provides a mechanism for iteration over all positions. A cursor might allow this as well, but it does not have to (like our earlier example of a stack ADT).

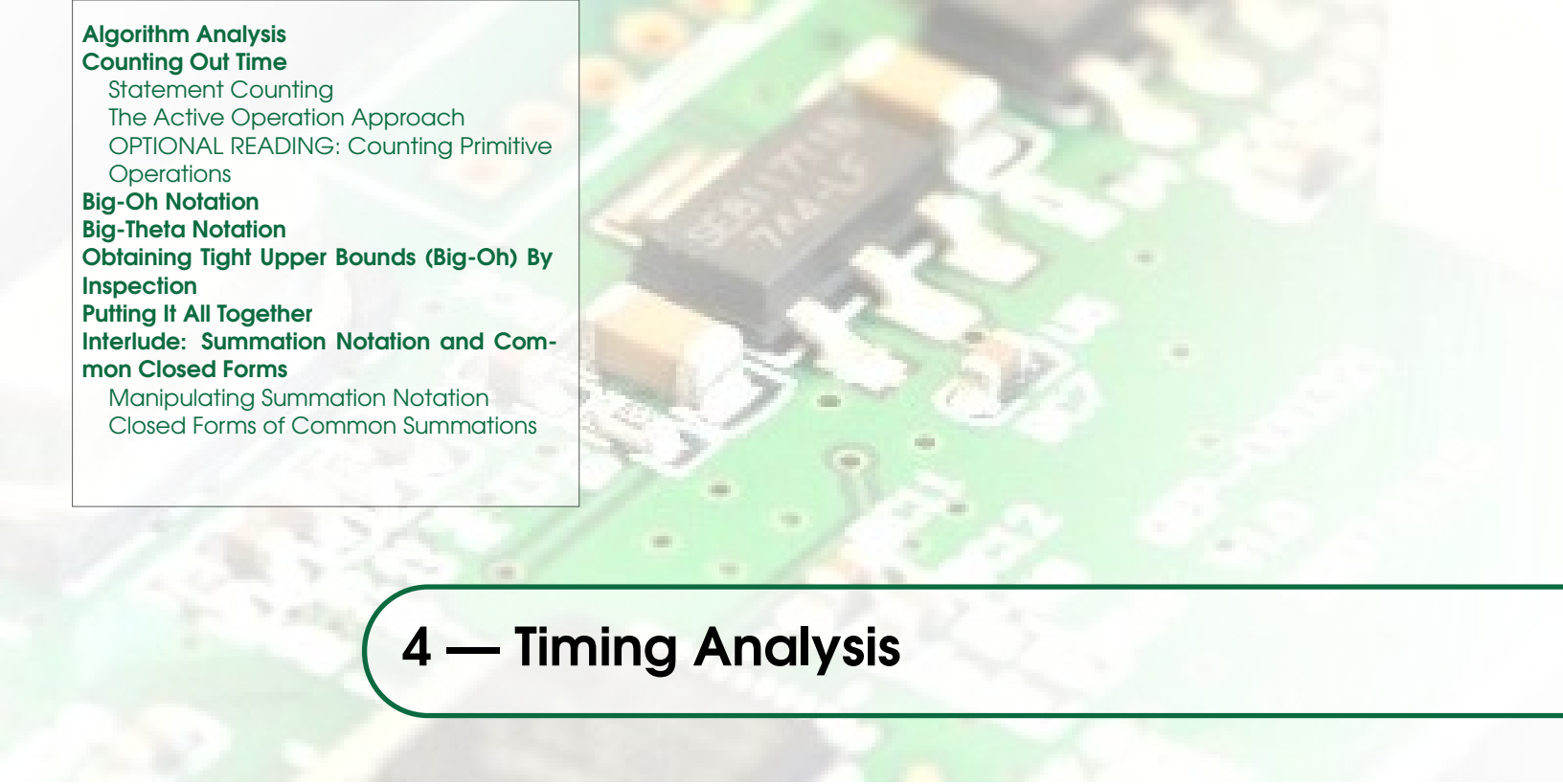
We could imagine that our list classes have a method called `iterator()` which returns an object that records a cursor position within a particular instance of a list. This iterator object would also implement the cursor interface from Section 3.3. So the idea behind iterators is this:

1. An instance method of a list (or any collection) is called that returns an iterator object.
2. The iterator object implements the cursor interface.
3. The iterator object can be used to "loop" over the elements of the list instance which generated it.

Why would we want to do this? Isn't it good enough for our data structures to have cursors? Well, it is useful to be able to create iterator objects from an instance of a collection when the cursor of the collection instance is being used for something else other than iterating over all its contents.

Otherwise, it would be up to the user to save and restore the collection's cursor before and after using the cursor to iterate. This is not desirable.

We will get into the details of implementing an iterator in the course of doing other things. For now, be content knowing that cursors are properties of data structures, and iterators are objects separate from the data structure instance in which they represent a position.



4 — Timing Analysis

Learning Objectives

After studying this chapter, a student should be able to:

- describe two different methods for counting the actual running time of an algorithm;
- intuitively explain the definitions of Big-Oh and Big-Theta notation and what they mean;
- construct proofs of the big-Oh notation of simple functions;
- state which of two standard growth functions grows more quickly; and
- informally simplify expressions describing running time of algorithms by inspection.

4.1 Algorithm Analysis

We can measure many different qualities of an algorithm: simplicity, readability, verifiability (is it correct?), validity (does it solve the desired problem?), modifiability/extensibility, reusability, portability (other machines, languages), and **efficiency**.

It is difficult to measure most of these qualities in a quantitative way, with the exception of efficiency. Efficiency of an algorithm can be measured for any resource that the algorithm consumes. The most commonly measured resource is time. If we are measuring time efficiency, we want to know how long an algorithms takes to run relative to the size of its input. Another fairly commonly measured resource is memory space. When we measure space efficiency, we want to know how much memory is required by the algorithm relative to the size of the input.

Although our focus in this course will be on the measurement of time and space requirements, other possible measures of efficiency, all measured relative to the size of the input, include:

- the number of processors/cores required by the algorithm;
- the number of threads required;
- the amount of network bandwidth used;
- the number of memory fetches; and
- the number of CPU cache misses.

4.2 Counting Out Time

You may recall from CMPT 115 that, ideally, we wish to consider the time and/or space requirements of an algorithm independent of the hardware on which it will be running, the language in which it will be implemented, and the runtime environment in which it will exist (e.g. system load, competition with other programs for resources). We would also like our analysis methods to be applicable to pseudocode algorithms, which they will be since they will be independent of programming language, and pseudocode is essentially just another programming language. We will see two methods of measuring time requirements which achieve these objectives.

We will focus on analysis of time requirements for now. In order to analyze time, we need to be able to count out the number of time steps that are used by an algorithm for a given input size N. There are a number of methods for doing so, which are ultimately all equivalent. Two methods are described below, the first of which you may have seen before.

4.2.1 Statement Counting

In the statement counting approach, we measure time by counting the number statements executed by the algorithm for an input of size n. In other words, we would like to find a function $T(n)$ such that $T(n)$ is the number of statements executed by the algorithm when the input is size n. We assume that each line of code takes approximately the same amount of time to execute. This is, of course, a completely unrealistic assumption. We will see later exactly why this unrealistic assumption still allows us to come to a good analysis of the time required by the algorithm, but for now we content ourselves by accepting that the variation in time required by different statements will be shown to be small enough to be irrelevant. The one exception to this is statements that call functions. Function calls must be analyzed separately, and the time required by the function calls added to that of the calling algorithm; but we'll come back to this idea later. For now, let's consider an example without function calls.

■ **Example 4.1** Consider the following program which counts the number of times the string `s` occurs in the array of strings `a`:

```
public static int count(String[] a, String s)
{
    int count = 0;
    int i = 0;
    while (i < a.length)
    {
        if (s.equals(a[i]))
            count = count + 1;
        i = i + 1;
    }
    return count;
}
```

How many statements are executed if the array `a` contains n strings? We want to find $T_{\text{count}}(n)$ which is the number of statements executed by the `count` function given an array `a` of length n. Let's start by considering the `while`-loop. When analyzing loops, it usually helps to ask the following questions:

1. How many statements are executed by one iteration of the loop?
2. How many iterations of the loop occur?

Then the number of statements executed by the **entire** loop is just the product of the answers to questions 1 and 2. So let's answer these questions.

The loop body executes either three or four statements in one iteration (this includes the while loop condition) depending on whether or not the condition in the `if`-statement is true. When faced with a situation like this, we usually assume the worst, that is, we assume that the `if`-statement is always true. We can do this even if it can never actually happen, and it won't affect the results.[1] So in the worst case, one loop iteration executes 4 statements. That answers question 1. Now, how many times does the loop execute? Well, `i` starts at 0, and the loop executes until `i` is greater than or equal to `a.length` (don't forget that `a.length` $= n$). So that means that the loop condition is true exactly n times for values of `i` between 0 and $n-1$, inclusive (the loop executes once for each element of the array a); this answers question 2. We now know that the loop executes n times, and executes 4 statements each time (in the worst case) for a total of $4n$ statements. Now we consider the statements outside the loop; these only happen once. There are the two assignment statements before the loop, and there is the return statement after the loop; that's three more statements, so we're up to a total of $4n+3$. But let's not forget that we only counted the number of times the loop condition was true. In order for the loop to stop, the loop condition must have become false at some point, in this case, when i got a value of `a.length` = n. So that's a grand total of $4n+4$ statements, so our final answer is $T_{\text{count}}(n) = 4n+4$. ∎

Note that we could also consider the best case scenario, which is when the `if`-statement is always false. In that case there would be 3 statements per loop iteration, n iterations, and four additional statements for a grand total of $T_{\text{count}}(n) = 3n+4$.

4.2.2 The Active Operation Approach

Our next approach for counting time works by identifying one statement in the algorithm that executes at least as often as any other, and is "central" to the algorithm. Such a statement is called the *active operation*. Being "central" to the algorithm is **very** important. So, what does it mean for a statement to be "central" to an algorithm? One way to think of it is as a statement that does the bulk of the work for the algorithm. Suppose there are two statements that execute equally often, and at least as often as any other statement. If one of these is an assignment statement, and one is a function call, then we would choose the function call as the active operation because it is more "central" to the algorithm; there is potentially a lot more work going on in that function call than the assignment statement. Generally, if there are no function calls, then it suffices to choose a statement that executes at least as often as any other.

Once we have selected the active operation, we then determine the exact number of times the active operation executes and this is our $T(n)$ for the algorithm. Let's repeat Example 4.1 using the active operation approach.

[1] The reason for this is the same as the reason why we can assume that all statements take the same amount of time — the difference is too small to matter.

■ **Example 4.2** Here is the same program as in Example 4.1. We need to select an active operation, and determine how many times it executes.

```
public static int count(String[ ] a, String s)
{
    int count = 0;
    int i = 0;
    while (i < a.length)
    {
        if (s.equals(a[i]))
            count = count + 1;
        i = i + 1;
    }
    return count;
}
```

An active operation is a statement that executes at least as often as any other. We know from our previous analysis that each statement in the `while`-loop body executes, at most, n times because the `while`-loop condition is true n times (where n = `a.length`). Is there a statement that executes more often? In fact, there is: the loop condition. The loop condition is true n times, and false one time; it executes $n+1$ times. There is no other statement that executes $n+1$ or more times, so the loop condition is our active operation. Since it executes $n+1$ times, then our final answer is $T_{\text{count}}(n) = n+1$. ■

It should be clear at this point, that once an active operation has been identified, computing $T(n)$ is much easier with this approach. But there are a couple of issues:

- What if it's not obvious which statement executes more than any other?
- Why didn't we get the same answer as with the statement counting approach?

The first question is easy to answer. If we aren't sure which of two statements executes more often, we can perform the active operation approach on both active operations. Simply find out exactly how many times each candidate operation executes, and choose the one that executes more often as the final answer. The second question is more difficult. We said that these two methods are equivalent, but at the moment, it would seem not to be the case. The answer will become clear when we reduce our $T_{\text{count}}(n)$ expressions into simpler functions using Big-Oh notation. The key thing to observe at this point is that the $T_{\text{count}}(n) = 4n+4$ that we got from statement counting and the $T_{\text{count}}(n) = n+1$ we got from the active operation approach are both linear functions. Both functions are $O(n)$. You may recall that multiplicative and additive constants are not at all important in timing analysis, which is why we use Big-Oh notation. Once we convert both $T_{\text{count}}(n)$ functions to Big-Oh notation (effectively ignoring the additive and multiplicative constants) we will get the same answer!

4.2.3 OPTIONAL READING: Counting Primitive Operations

You may have seen this method of time analysis used in CMPT 115. A *primitive operation* is an operation that we regard as "indivisible," at some level of abstraction. Examples of primitive operations are arithmetic operations (+, -, $\times$, /), logical operations (and, or, not, etc.), relational operators ($<$, $>$, $\leq$, equality, etc.), and assignment of a value to a variable.

Using this approach, we count the number of primitive operations executed by the algorithm with an input of size n and use that as our $T(n)$. This approach, ultimately, is equivalent to the other two, above. While it perhaps more realistically represents how many actual operations are executed in an

algorithm, it is very cumbersome because there are a lot more primitive operations than statements (or active operations). Counting primitive operations usually results in a much more complicated $T(n)$, but which, when reduced using Big-Oh notation, ends up giving us the same answer as the other approaches anyway. This is why we prefer the above two approaches: they are usually simpler and amount to less work.

4.3 Big-Oh Notation

Big-Oh notation is a concise mathematical notation for *upper bounds* on functions. We're going to define Big-Oh notation more formally and rigorously than you've seen it before. This may seem unnecessary as you might already have a good intuition about Big-Oh, but it is very important for actually proving that algorithms have a certain efficiency; and hey, you're in second year now, so it's time for the real deal. Let's do this...

Let $f(n)$ be a function (e.g. n, n^2, 2^n, etc.). Intuitively, $O(f(n))$ is defined as the *set* of functions that grow no faster $f(n)$ (up to a constant factor). The idea that $O(n)$, $O(n^2)$, $O(2^n)$, etc., are **sets** is probably a new one for you. But mathematically, this is the case.

Definition 4.1 Let $f(n) : \mathbb{Z}^+ \to \mathbb{R}^+$ be a function from positive integers to positive real numbers.

$$O(f(n)) = \{t(n) \mid \exists c, n_0 > 0 \text{ such that } t(n) \leq cf(n), \forall n \geq n_0\}$$

Let's break this down: First we can see that $O(f(n))$ is a set; note the curly brackets used for the set notation. This set is defined as all of the functions $t(n)$ which satisfy some conditions. So what are those conditions? First let's remember what those funny symbols mean — you should have come across them in CMPT 260 if you've taken it. The symbol $\exists$ is read as "there exists" and the symbol $\forall$ is read as "for all". Thus, the conditions that $t(n)$ must satisfy to be in the set say that there must exist two constants, one called c, and one called n_0 (which may be different values for different $t(n)$'s) , such that the statement $t(n) \leq cf(n)$ is true for every value of n greater than or equal to n_0. And that's it. That's Big-Oh notation and it's no more complicated than that.

The c is the part that makes multiplicative constants irrelevant. If we have the function $t(n) = 4n$, then we can say that this function is in the set $O(n)$ because if we choose c to be equal to, say, 10, and choose n_0 to be equal to 1, then $(t(n) = 4n) \leq (cf(n) = 10n)$ for all $n > n_0 = 1$, which meets the conditions given for $t(n)$ to be in the set $O(n)$. Because there was a c and an n_0 for which the conditions were true, $t(n) = 4n$ is $O(n)$. It works the same way for any $f(n)$ and $t(n)$. This is also why our incorrect assumption that all statements take the same amount of time doesn't cause any problems. It doesn't matter if a statement actually takes half as long or twice as long as another; the fact that multiplicative constants are irrelevant when $T_{\text{count}}(n)$ for the algorithm is expressed in Big-Oh notation also makes variations in statement execution time irrelevant.

You may not realize it, but in the previous paragraph, we did a proof! We mathematically proved using the definition of Big-Oh that the function $4n$ was in $O(n)$, the set of linear functions. Let's do another proof. Let's prove that $3n^2 + 2$ is in the set $O(n^2)$.

■ **Example 4.3** Prove that $3n^2 + 2$ is $O(n^2)$.

To do this, we have to show that there are numbers c and n_0 such that $3n^2 + 2 \leq cn^2$ for all $n \geq n_0$. It's easy to see that $2 \leq n^2$ for all $n \geq 2$. Now add $3n^2$ to both sides and we get the inequality $3n^2 + 2 \leq 4n^2$ for all $n \geq 2$; so we now know this inequality to be true. Now if we go back to what

we wanted to prove,

$$\text{there exists a } c \text{ and an } n_0 \text{ such that } 3n^2+2 \leq cn^2 \text{ for all } n \geq n_0$$

and substitute $c = 4$ and $n_0 = 2$, we get:

$$3n^2+2 \leq 4n^2 \text{ for all } n \geq 2$$

which we know is true! So there *does* exist a c and n_0 that makes the condition true, and therefore $3n^2+2$ is $O(n^2)$. ∎

We'll do more examples in class for practice.

4.4 Big-Theta Notation

In the previous section we proved that the function $t(n) = 3n^2+2$ was $O(n^2)$. The thing is, $t(n)$ is **also** $O(n^3)$! We showed this by proving that $3n^2+2 \leq 4n^2$ for all $n \geq 2$. It must also be true, then, that $3n^2+2 \leq 4n^3$ for all $n \geq 2$, which proves that $3n^2+2$ is $O(n^3)$. It makes sense because a function that grows no faster than n^2 also cannot grow faster than n^3. For that matter, a function that grows no faster than n^2 cannot grow faster than 2^n, so our function $t(n)$ is also $O(2^n)$. It's also $O(n!)$. More generally (remembering that Big-Oh notation is actually notation for sets!):

$$O(1) \subset O(\log n) \subset O(\sqrt{n}) \subset O(n) \subset O(n^2) \subset \ldots \subset O(n^k) \subset O(2^n) \subset O(n!)$$

Knowing that a function is $O(n!)$ isn't very helpful, if it is also $O(\log n)$. What we are **really** interested in are what we call *tight* upper bounds. That is, we want to know the least quickly growing function $f(n)$ such that our algorithm's time function $t(n)$ is still $O(f(n))$.

Thus, it would be nice if we had another notation to capture this intuition. Big-Theta notation captures this. Intuitively, we say that $t(n)$ is $\Theta(f(n))$ if $t(n)$ is **exactly** $f(n)$ (within a constant factor).

Definition 4.2 Let $f(n) : \mathbb{Z}^+ \to \mathbb{R}^+$ be a function from positive integers to positive real numbers.

$$\Theta(f(n)) = \{t(n) \mid \exists b, c, n_0 > 0 \text{ such that } bf(n) \leq t(n) \leq cf(n), \forall n \geq n_0\}$$

The definition is very similar to Big-Oh. First we note that $\Theta(f(n))$ is, like $O(f(n))$, a set. It is the set of functions $t(n)$ that meet the given conditions. In fact it is the same definition as $O(f(n))$ except that we have an additional constant b, and an additional condition involving b. We must have that $t(n) \leq cf(n)$ (just like Big-Oh!) **and** that $bf(n) \leq t(n)$ for all $n \geq n_0$. To put it another way, $t(n)$ must grow both at least as quickly and at most as quickly as $f(n)$ (within a constant factor).

We can say that $t(n) = 3n^2+2$ is $\Theta(n^2)$ because in addition to $3n^2+2 \leq 4n^2$ for all $n \geq 2$, we can also show that $0.5n^2 \leq 3n^2+2$ for all $n \geq 2$. Thus for $b = 0.5$ and $c = 4$, and $n_0 = 2$, the condition holds, which proves that $t(n) = 3n^2+2$ is $\Theta(n^2)$.

4.5 Obtaining Tight Upper Bounds (Big-Oh) By Inspection

Suppose we have successfully applied either the statement counting approach or active operation approach, and we have arrived at a function $t(n)$ which tells us how much time our algorithm

requires for an input of size n. The expression could have several terms, like: $t(n) = 1.5n^2 + 4.5n - 4$. Performing mathematical proofs every time for expressions like this would be, admittedly, rather cumbersome. It turns out that for most simple algorithms, we can just choose the term in our $t(n)$ that we know grows most quickly (e.g. because we know that quadratic functions always grow more quickly than linear ones, or that linear functions always grow more quickly than logarithmic functions), and strip off the multiplicative constants. Thus we can obtain the Big-Oh notation for an expression by inspection. This technique gives us not only an upper bound, but a *tight* upper bound. For example:

Expression	Most quickly growing term	Big-Oh (tight bound)
$2n^3 + 5n + 1$	$2n^3$	$O(n^3)$
$5n + \sqrt{n} + 50$	$5n$	$O(n)$
$5n + 12n\log n + 10000 + \frac{1}{10}\log n$	$12n\log n$	$O(n\log n)$
$1000000n^2 + n^1.5 + 16n + \frac{1}{100}2^n$	$\frac{1}{100}2^n$	$O(2^n)$

The typical simple functions used in timing analysis are listed below in order from least quickly growing to most quickly growing.

- 1
- $\log n$
- $\sqrt{n} = n^{\frac{1}{2}}$
- n
- $n\log n$
- n^2
- $n^2\log n$
- n^3
- $\vdots$
- n^k
- n^{k+1}
- $\vdots$
- 2^n
- 3^n
- $\vdots$
- a^n
- $(a+1)^n$
- $n!$

4.6 Putting It All Together

Earlier in this chapter, we analyzed the `count` algorithm which counted the number of times a string appeared in an array of strings. We used the statement counting approach, and discovered that, in the worst case, $t(n) = 4n + 4$ operations were required. We also used the active operation approach, which told us that the active operation executed $t(n) = n + 1$ times. Whichever approach we use, the next step is normally to take the resulting time function and express it in Big-Oh or Big-Theta notation. We can do this using the "informal" inspection method from the previous section. For $4n + 4$ the most quickly growing term is $4n$ which means it is $O(n)$. For $n + 1$, the most quickly

growing term is n, which is also $O(n)$. Thus, once we simplify the expression to Big-Oh notation, both approaches give us the same result. If used properly, both statement counting and the active operation approach (and, if you read the optional section, counting primitive operations) should always give the same Big-Oh result precisely because constants are ignored when converting to Big-Oh notation.

So where does Big-Theta come into all of this? Recall how we performed both a worst-case and a best-case analysis of the `count` method using the statement counting approach. The results were $4n+4$ in the worst case, and $3n+4$ in the best case. Both of these expressions are $O(n)$ — this means that the `count` method is also $\Theta(n)$. In general, if one performs a best-case and a worst-case analysis, and the results, in Big-Oh notation, are the same, that is, both are $O(f(n))$, then the algorithm is also $\Theta(f(n))$. If, however, the best case and worst case runtimes (in Big-Oh notation) are different, then there is **no** function $f(n)$ for which the algorithm is $\Theta(f(n))$.

Big-Theta is a **stronger** statement than Big-Oh. Big-Oh says that the algorithm's runtime as a function of input size is no worse than (and could even be better than) $f(n)$, but Big-Theta says that the algorithm's runtime as a function of input size is **exactly** $f(n)$ (up to a constant factor).

4.7 Interlude: Summation Notation and Common Closed Forms

In timing analysis, long sums with many terms often arise for which we use summation notation:

$$\sum_{i=1}^{n} x_i = x_1 + x_2 + \cdots + x_n$$

Here we review how to manipulate expressions with summations in them, and a few common summations and their closed forms. Doing so will aid in our manipulation and simplification of the mathematical expressions that arise from timing analysis. While these summations did not arise from any of the examples in this chapter, they will come up in class, and later in this book, so we may as well learn them now.

4.7.1 Manipulating Summation Notation

A sum can be broken up by separating its index set, for example:

$$\sum_{i=1}^{n} x_i = \sum_{i=1}^{k-1} x_i + \sum_{j=k}^{n} x_j,\ 1 < k \leq n.$$

Above, we see how the sum for $i = 1$ to n can be separated into a sum for $i = 1$ to $k-1$ and another sum for $i = k$ to n, for any k between 1 and n.

If the body of a sum contains additive (or subtractive) terms, these can be separated by distributing the summation notation over each term:

$$\begin{aligned}\sum_{i=1}^{n}(2x_i+3y) &= \sum_{i=1}^{n}2x_i + \sum_{i=1}^{n}3y \\ &= 2\sum_{i=1}^{n}x_i + 3\sum_{i=1}^{n}y\end{aligned}$$

Note how the second line also demonstrates how multiplicative constants can be moved outside the summation.

4.7.2 Closed Forms of Common Summations

The following summations often arise from the timing analysis of loops. You must know the closed forms of these sums because this is crucial for simplifying the expressions in which the summations arise. All of the following identities can be proved by induction (but we won't!).

$$\sum_{i=1}^{n} i = 1+2+\cdots+n = \frac{n(n+1)}{2} \qquad \text{(sum of first } n \text{ integers)}$$

$$\sum_{i=1}^{n} i^2 = 1+4+9+\cdots+n^2 = \frac{n(n+1)(2n+1)}{6} \qquad \text{(sum of first } n \text{ squared integers)}$$

$$\sum_{i=0}^{n} a^i = \frac{(a^{n+1}-1)}{(a-1)} \text{ for } a \neq 1 \qquad \text{(sum of first } n+1 \text{ non-negative integer powers of } a\text{)}$$

$$\sum_{i=1}^{n} i a^i = \frac{n a^{n+1}}{(a-1)} + \frac{a(a^n-1)}{(a-1)^2)} \text{ for } a \neq 1$$

The first two identities, above, are particularly important as they very often arise from nested loops. The special case of the third identity where $a = 2$ tends to come up a lot as well, especially when looking at binary trees; this would be the familiar sum $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$. The fourth one is rarer, but still a good one to know!

5 — Abstract Data Types

Learning Objectives

After studying this chapter, a student should be able to:

- give a definition of a *data structure*;
- give a definition of an *abstract data type*;
- distinguish the familiar term *data type* from the previous two terms;
- explain the advantages of using ADTs and encapsulation; and
- write programming-language-independent specifications of basic ADTs using the specification method discussed in Section 5.3.

5.1 Data Structures

Lists, arrays, stacks, queues, and trees are all examples of data structures. On the surface they don't look like they have a lot in common, but they do have some shared features: they are all collections of data elements with defined relationships. It is from this that we get the definition of a data structure.

Definition 5.1 A *data structure* consists of:
1. a collection of data elements; and
2. a set of associations or relationships (i.e. structure) between elements.

In a list, the data elements are usually all of the same kind, for example, integers, real numbers, character strings, or even a compound data type. In a list, the relationship between elements (the structure) is that each element has a rank in a linear ordering of elements.

In a tree, the elements are, again, usually all of the same kind (e.g. integers, character strings, etc.), and the relationship between elements is a hierarchical one: each element has a parent, and zero or more children.

5.1.1 Data Type vs Data Structure

The term *data structure*, which we have just defined, is often confused with another term: *data type*. A *data type* is nothing more than the description of a **kind** of data (e.g. integer, real, character string), that is, the manner in which bits and bytes of memory are interpreted. A data type does not consist of a collection of elements, and therefore cannot have structure or relationships between elements. On the other hand, each of the elements of data that make up a data structure have a data type.

5.2 Abstract Data Types

An abstract data type, or ADT, is an *abstraction* that permits programmers to make use of a data structure without knowing its internal implementation. A programmer knows **what** an ADT can do, but **how** it is done is hidden.

Definition 5.2 An abstract data type consists of:
- one or more data structures (i.e. a declaration of data);
- a set of publicly known operations on the data structure(s); and
- encapsulation (hiding) of the data structures and the implementation of the operations.

Let's look at a few examples of ADTs and take a close look at the three different components that comprise them.

■ **Example 5.1** A string ADT consists of:
the data structure: an array, or list of characters (users of the string ADT don't have to care which!)
the operations (interface): create, compare, concatenate, etc.
the encapsulation: implementation of the operations and underlying data structure are hidden in a library (e.g. a Java `String` object).

■

■ **Example 5.2** A list ADT consists of:
the data structure: an array, or singly linked list, or doubly linked list, etc.
the operations (interface): add at beginning, add at end, find element in list, delete n-th element, delete element equal to e, etc.
the encapsulation: implementations of the operations and the choice of data structure are hidden in the software implementation.

■

Even an integer can be viewed as an abstract data type (at a certain level of abstraction). Usually we view integers as primitive data since the encapsulation happens in hardware rather than software, but we can view an integer as an ADT as described in the next example.

■ **Example 5.3** An integer ADT consists of:
the data structure: a sequence of bits — might be 1's complement representation, 2's complement representation, signed magnitude representation; byte order might be big endian, little endian... doesn't matter. The programmer generally need not know because the hidden implementations of the operations take care of everything.
the operations (interface): $+, -, /, \times, \ldots, \%, \leq, \geq$, ==, etc.
the encapsulation: details of operations and choice of binary representation are hidden in hardware.

■

5.2.1 Why do we use ADTs?

We use ADTs because they specify the expected behaviour of a data type before it is implemented. This facilitates the independent development and implementation of different abstract data types by different programmers, and the development of reusable data types and objects by virtue of the public, and well-documented interface. Changes and improvements to the implementation of an ADT can be made without affecting other parts of the program as long as the public interface and expected behaviour of the operations does not change. Therefore it is crucial to design the interface well the first time, before the ADT is implemented and used.

Another advantage to using ADTs is that it postpones the development of details that are initially unnecessary. For example, suppose a program requires a tree data structure. When designing the various components of the program, it probably isn't that important to know whether the children of a node are stored in a linked list or an array. But the tree will have an agreed upon public interface which will allow the rest of the program to be designed, while the choice of implementation can be delayed until later.

Finally, ADTs allow complex data types to be designed independently from any particular programming language precisely because the implementation is encapsulated. If we specify the public interface in a programming-language-independent fashion, then we can delay the choice of programming language until much of the design process is complete, and we have a better idea of which programming language would be most suitable. But we can still work with the ADT and even design algorithms using it because we can find out **what** it will be able to do when it is implemented by consulting the public interface.

5.2.2 Why do we encapsulate the implementation of an ADT?

There is a concept in software engineering called *coupling*. Coupling refers to the degree to which two modules of code depend on each other. If we didn't use ADTs and allowed module A to use a singly linked list data structure directly, the coupling is higher because if the linked list is later improved to a doubly-linked list data structure, then module A would likely have to change as well. If we used an ADT instead and encapsulated the data structure used by the list, then module A just knows it is using a list, and the list implementation can change without affecting the code for module A. This is an example of lower coupling. Moreover, module A cannot inadvertently make illegal changes to the list data structure because it can't access the data structure directly, it can only access it through the public interface. Then, so long as the implementation of the operations are correct, the data structure will always be in a valid state. Thus, the use of ADTs reduces software coupling, which is a very good thing.

A related concept is *cohesion*. Cohesion refers to the degree to which the components of a module of code are related or belong together. Higher cohesion is desirable. ADTs generally lead to code that has higher cohesion because an ADT is usually implemented as a single module of code, and since all of the code is related to the ADT, all of the code in the module is closely related to a single purpose: implementing the ADT.

In short, use of ADTs enforces encapsulation, which leads to code that has lower coupling and higher cohesion.

5.3 ADT Specification

On a couple of occasions in this chapter we have mentioned or alluded to the idea of *specifying* the operations and functionality of an ADT in a programming-language-independent way. Specification

is distinct from implementation.

Specification is like a contract between the ADT and the users of the ADT. The specification outlines the expected inputs, promised outputs, and semantics of each ADT operation. It does **not**, however, make any promises about **how** the operations are carried out. It only promises that, for any implementation of the ADT, certain things will happen and/or be returned when a particular operation is invoked.

It is the hidden (encapsulated) implementation of an ADT that determines **how** the ADTs operations are performed. Choices made about the programming language, the particular data types used to represent data, the choice of data structures and the specific algorithms used to carry out the operations are not decided until implementation time, and the specification should not dictate any of these choices.

In this section we will examine **one** way of specifying ADTs. The formal methods of software specification are as many and varied as programming languages. The one we will examine is chosen because it is not **too** formal, and because it results in specifications that are easy to implement in Java, while still being independent of Java.

5.3.1 Set-based Specification

The only formalisms that we will use in our specification is the notion of sets, and the mathematical notation for specifying functions.

Sets will represent things like the set of all instances of the ADT, or the set of all integers. Sets are denoted by upper-case symbols. Suppose we wish to specify a list ADT. We might use the symbol L to represent the set of all lists, and the symbol G to represent the set of elements that can appear in a list. We use lower-case symbols, like l to represent a specific instance of a list, or g to represent a specific element that can appear in a list. We might also make use of the standard symbols for sets of numbers like the natural numbers ($\mathbb{N}$), the integers ($\mathbb{Z}$), or the real numbers ($\mathbb{R}$).

The mathematical notation for functions should be somewhat familiar to you. For example:

$$f : \mathbb{N} \to \mathbb{Z}.$$

This is, of course, the specification of a function, f, from the natural numbers to the integers. We will use a similar notation for specifying the signatures of the operations in our ADT specifications.

5.3.2 Components of the Specification

A specification will be comprised of five things:

- The name of the ADT.
- A definition of the sets used in the specification of the ADT.
- The signatures of the operations in the ADT, specified as functions defined on sets.
- The preconditions for each operation.
- The semantics of each operation.

A complete specification of a list ADT is given in Figure 5.1. We will now go through all of the components in this example and learn how to read and write them.

5.3.3 Specifying the Sets used in the Specification

To specify the sets, we just list the symbol used for the set, and say what it represents.

Name: List$\langle G \rangle$

Sets:
L : set of lists containing items from G
G : set of items that can be in the list
B : {**true**, **false**}

Signatures:
newList<G> : $\rightarrow L$
L.isEmpty: $\rightarrow B$
L.insertFirst(g): $G \rightarrow L$
L.firstItem: $\not\rightarrow G$
L.deleteFirst: $\not\rightarrow L$

Preconditions: For all $l \in L$, $g \in G$
newList<G>: none
l.isEmpty: none
l.insertFirst(g): none
l.firstItem: l is not empty
l.deleteFirst: l is not empty

Semantics: For $l \in L$, $g \in G$
newList<G>: construct new empty list to hold elements from G.
l.isEmpty: return true if l is empty, false otherwise
l.insertFirst(g): g is added to l at the beginning.
l.firstItem: returns the first item in l.
l.deleteFirst: removes the first item in l.

Figure 5.1: A List ADT Specification

In the list ADT in Figure 5.1 we have three sets. First we have the set L which is the set of all instances of the ADT we are specifying. Every ADT specification will define a set of all instances of the ADT because it is needed for the specification of signatures, preconditions, and semantics. We also see a set G of all the items that are permitted to be in the list; note that we don't say specifically what they are because this maintains the generality of the specification. We can decide later at implementation time what elements actually are members of G. Note that every specification of an ADT which is a *collection* of elements will include the definition of the set of items that may be in the collection. The final set in the list ADT is B which consists of the Boolean values **true** and **false**. We need this for specifying operations that return a boolean value.

5.3.4 Specifying the Name of the ADT

You can name the ADT whatever you like. The only thing we need to note here is the convention of putting the set G in angle brackets after the name of the ADT. This is a convention used to make it clear at a glance that a List is a collection of elements from the set G. Although this notation resembles the syntax for generic types in Java, this is just a coincidence; the latter has nothing to do with the former.

5.3.5 Specifying the Signatures of Operations

Signatures are specified using the same notation used to specify functions in math, which we reviewed in Section 5.3.1. We use the same notation for our data structure specification.

In the list ADT in Figure 5.1 we see that our list has signatures specified for five operations. Let's start with the signature for insertFirst. The "L." at the beginning designates that this is an operation on elements of the set L, i.e. an operation on a list. The $G \rightarrow L$ part indicates that the operation defines a function from the set G to the set L. This is normally interpreted to mean that the function takes an element of G as input and returns an element of L. Now intuitively we know that insertFirst doesn't actually return a list, it just changes the state of an existing list. Since the L after the arrow matches the L before the ".", this is interpreted as a change of state of the list to which the operation

is applied, rather than actually returning something. If the set after the arrow does not match the set before the ".", then the operation is actually returning something (we'll see this in a moment!). Finally, the "(g)" is giving a name to the element from G that is a parameter to this operation. It is implicit in the notation that g is an element of G because G is the only set appearing before the arrow.

Now let's consider the signature for isEmpty in Figure 5.1. The signature starts with "L.", so we know this is an operation on a list. The symbol B after the arrow does not match the L before the "." so we know that this operation returns an element of B, i.e. either **true** or **false**. But there is no set symbol **before** the arrow — what does this mean? It means that the operation does not take any parameters, it just returns a result. This makes sense because intuitively we know that the isEmpty operation should take no parameters and return a Boolean value indicating whether the list is empty — but note that the signature does not actually specify **any** of these semantics. It merely specifies the sets which from which we can draw valid inputs and outputs. The semantics come later.

The signature for the firstItem operation illustrates something new yet again. This signature very closely resembles the signature for isEmpty — it takes no parameters and returns something from the set G — but the arrow has a line through it. This indicates that the function is a *partial function* which means that it is not defined for all possible inputs. Intuitively, we know that firstItem cannot be used on an empty list because there is no element to return, but that is not specified here. This is instead captured in the section defining the preconditions for the operations. Generally, if an operation has a precondition, its signature has the line through the arrow to indicate that it is not defined for all possible inputs or instances of the ADT.

The deleteFirst signature looks very much like the firstItem signature except that it just changes the state of the instance to which it is applied rather than returning anything — in fact it takes no parameters and returns nothing. Because the L after the arrow matches the L before the ".", we know that this operation just changes the state of the list from L to which it is applied. This is again consistent with our intuition that deleteFirst removes an item from a list, thereby changing the state of the list.

Finally, we consider the newList$\langle G \rangle$ operation. This signature does not start with "L." because it is not applied to an element from L, but rather **creates** an element of L. This idea is further reinforced by the L appearing after the arrow. This indicates that the operation returns an element of L (since it doesn't match what comes before the ".").

5.3.6 Specifying the Preconditions

For operations that may not be defined for all inputs, we need to say what conditions must hold before the operation can be applied. First let's observe in the list of each operation's preconditions we now have a lower-case "l" before the ".". It is written like this because now we are talking about the operations applied to a specific instance l from L, and any preconditions we list must be met by the specific instance l. If an operation has no preconditions, we simply say so, as for the isEmpty operation. The precondition for firstItem reads like this: "firstItem can be applied to the list l only if l is not empty.".

Note how the "(g)" in brackets appears with the insertFirst operation in the list of preconditions. This is included because it might be the case that a precondition applies to a parameter of an operation.

Finally, note that the list of preconditions applies to all $l \in L$ and all $g \in G$ — this is specified at the top of the preconditions section. This means that the given preconditions apply all the time, to

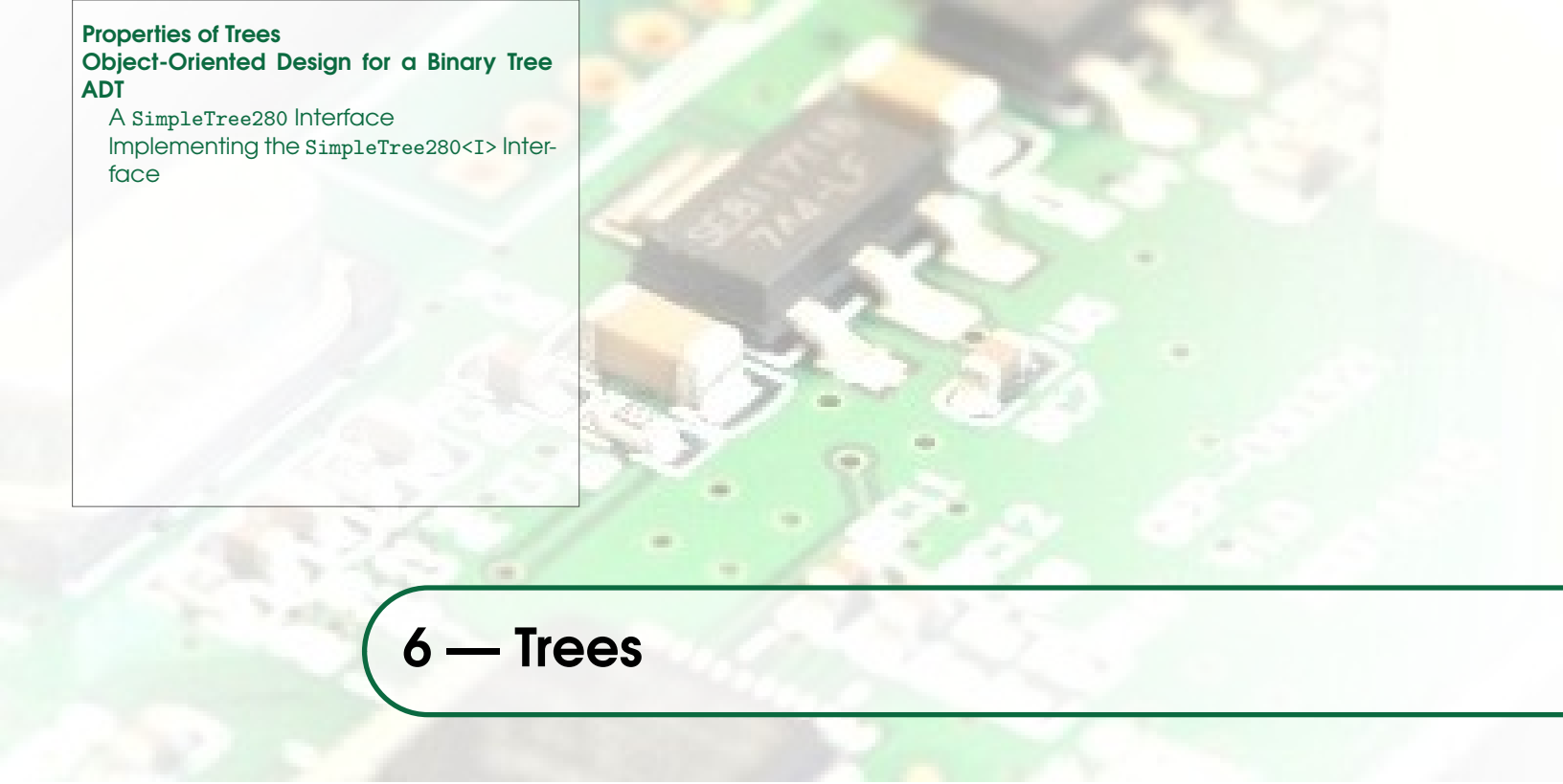
any list l from the set L and any element from the set g.

5.3.7 Specifying the Semantics

The syntax for the Semantics section of the specification is largely the same as for the Preconditions section except that here we actually say *what* each operation is supposed to do, making reference to the instance the operation is applied to (in this case, l) and possibly to the parameters of the operations, for example, the g in insertFirst.

5.3.8 More Examples?

We will go through more examples of specifications, and practice writing specifications during the in-class exercises and on assignments. If you want to practice now, try to write a specification for an ADT with which you are familiar from CMPT 115, such as a stack or a queue.



6 — Trees

Learning Objectives

After studying this chapter, a student should be able to:

- define the terms *parent, child, ancestor, descendent, sibling, level, and height* with respect to trees;
- explain the difference between leaf nodes and internal nodes;
- define what a binary tree is;
- identify the information that must be stored in a binary tree node; and
- informally describe how one might implement a binary tree within the framework of `lib280`.

6.1 Properties of Trees

Arrays, lists, stacks and queues are all linear structures. Each of these ADTs impose a total ordering on the collection of elements they contain. Trees, on the other hand, allow the collection of elements they contain to be organized hierarchically. We normally think of a tree as containing nodes which are connected by arcs or edges (visualized as a straight line). Each node of the tree contains one element in the collection.

Each node in a tree has one *parent* and zero or more *children*, except for one special node called the root node that has no parent. Figure 6.1 shows three examples of trees. The root node of the bottom-right tree contains the element 4. As required, this root node has no parent, but it has two children (nodes containing 2 and 12). The node containing 12 is the parent of the node containing 7 (because 7 is the child of 12).

An element in a tree cannot be said to be "before" or "after" another one. Elements in a tree have more complex relationships, one of which is the parent/child relationship. Other relationships are *ancestor*, *descendent*, and *sibling* which we now define, in turn.

For any node s in a tree, there is only one path back to the root that does not repeat any edges or vertices. A node t is an *ancestor* of a node s if t lies along this path. Or more formally:

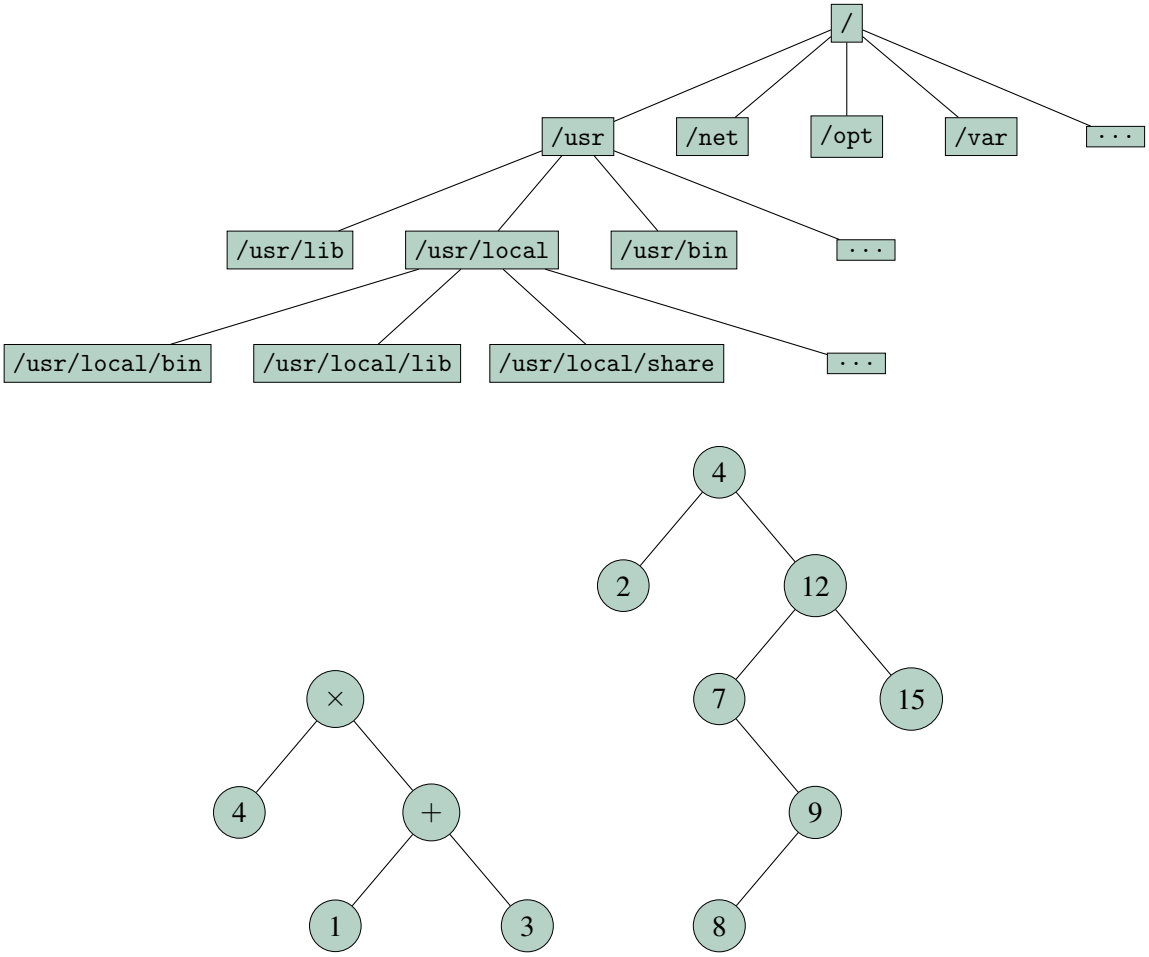


Figure 6.1: Three examples of a tree. Top: a tree representing a filesystem directory hierarchy; bottom left: an expression tree representing the mathematical expression $4 \times (1 + 3)$; bottom right: a binary search tree.

Definition 6.1 A node t of a tree is an ancestor of a node s if:
1. t is the parent of s; or
2. t is the parent of an ancestor of s.

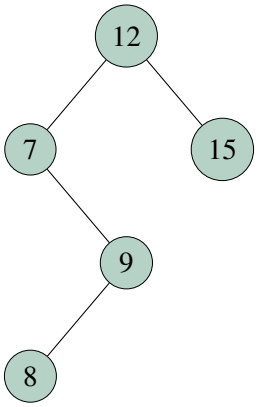
This is an example of a recursive definition. The first part of the definition is the base case: it tells us that the parent of s is an ancestor of s. The second part of the definition is the recursive part which tells us that if we already know that some node p is an ancestor of s, then the parent of p is also an ancestor of s. In this way we can discover all the ancestors of s. In the bottom-right tree of Figure 6.1, the nodes containing 7, 12, and 4 are all ancestors of the node containing 9, but 15 is not an ancestor of 9 because it is not on the path from 9 back to the root.

The opposite of an ancestor is a *descendent*. A node s is the descendent of a node t if t is an ancestor of s. Thus, in the bottom-right tree of Figure 6.1, the nodes containing 7, 8, 9, and 15, are all descendants of 12, because they all have 12 as an ancestor.

If two nodes have the same parent then they are *siblings*. In the bottom-right tree of Figure 6.1, 2 and 12 are siblings, as are 7 and 15. The node containing 9, however, has no siblings.

There are a number of ways to count the height of a tree, but to reduce confusion, we will choose one and keep consistent throughout the course. The *level* of a node t is the length of the path from t to the root. The root of a tree is on level 0; the children of the root node are on level 1, etc.. The number of different levels in a tree is its *height*. A tree consisting of just a root node has height 1. An empty tree has height 0. The bottom-right tree of Figure 6.1 has nodes on 5 different levels, thus it has height 5. The node containing 7 is on level 2 of the tree because there are two arcs on the path from 7 to the root.

The *subtree rooted at node t* is the tree formed by only the node t and all of its descendants. For example, the subtree rooted at node 12 of the bottom-right tree in Figure 6.1 is this:



A node that has no children is called a *leaf node* (or sometimes *terminal node*). Nodes that have at least one child are *internal nodes* (or *non-leaf nodes*). For example, in bottom-right tree in Figure 6.1, the nodes containing 2, 8, and 15 are leaf nodes because they have no children; the remaining nodes are internal nodes.

6.2 Object-Oriented Design for a Binary Tree ADT

In this section we will sketch the object-oriented design of a binary tree. You already know that a binary tree is a tree in which each node has between zero and two children. Our design will be based on a recursive definition of binary trees.

Name: SimpleTree$\langle G \rangle$

Sets:
T : set of trees containing elements from G
G : set of elements allowed in the trees
B : {**true**, **false**}

Signatures:
newTree$\langle G \rangle$: $\rightarrow T$
T.initialize(t_1, g, t_2): $T \times G \times T \rightarrow T$
T.isEmpty: $\rightarrow B$
T.rootItem: $\not\rightarrow G$
T.rootLeftSubtree: $\not\rightarrow T$
T.rootRightSubtree: $\not\rightarrow T$

Preconditions: For all $t \in T$
t.rootItem: t is not empty
t.rootLeftSubtree: t is not empty
t.rootRightSubtree: t is not empty

Semantics: For $t, t_1, t_2 \in T$, $g \in G$
newTree$\langle G \rangle$: construct a new empty tree to hold elements from G.
t.initialize(t_1, g, t_2): initialize t to have root g, left subtree t_1 and right subtree t_2
t.isEmpty: return true if t is empty, false otherwise
t.rootItem: returns the root element of t.
t.rootLeftSubtree: returns the left subtree of t.
t.rootRightSubtree: returns the right subtree of t.

Figure 6.2: A specification for a simple binary tree.

Definition 6.2 A binary tree is either:
- Empty; or
- consists of a root node, a left (sub)tree and a right (sub)tree.

A tree with a single root node fits this definition. It consists of a root element, an empty left subtree (which, by our definition is a tree), and an empty right subtree. Now that we know that a single node is a valid tree, we can build a tree with a root node, a left subtree consisting of a single node, and a right subtree consisting of a single node, producing a tree with three nodes — a root, a left child, and a right child, which, by our definition, is a valid tree. We can now use this new 3-node tree to build even bigger trees, according to the definition.

So for a basic tree design that fits our recursive definition, we would need a tree object that:
- can be initialized as an empty tree;
- can be initialized with a root, left subtree, and right subtree;
- report whether it is empty or not;
- can provide the data item in its root node; and
- can provide a reference to its left and right subtrees;

This will allow us to both build trees from smaller trees, much as we did in the preceding paragraph, and to query the tree to find out what data it contains. That said, this is still a **very** basic tree without much functionality. The specification for a tree that can do all of these things is provided for reference in Figure 6.2.

We would like to demonstrate how to implement such a tree within the `lib280` data structure library. The first thing we will do is convert the signatures section of the specification in Figure 6.2 into a java interface.

Since a tree contains a collection of elements, it fits the requirements of a `lib280` container. The `Container280` interface in `lib280` defines the methods that any container class in `280` must implement. The UML diagram for the `Container280` class is given in the top portion of Figure 6.3. In lib280, all containers must be able to report whether they are empty or not, full or not, and must

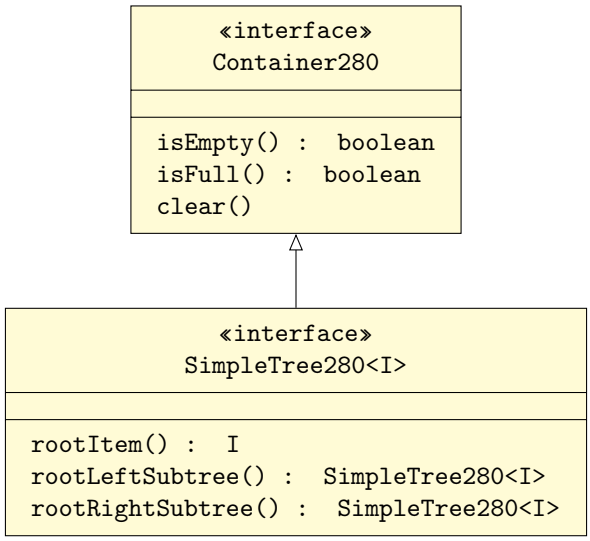


Figure 6.3: The `Container280` class and its extension to a `SimpleTree280<I>`.

implement a `clear` method that removes all elements from the container.

6.2.1 A `SimpleTree280` Interface

Since our simple tree interface is also to be a container, its interface must extend the `Container280` interface. Thus, it will contain all of the methods in `Container280`. To complete our simple tree interface, we add definitions for the methods `rootItem`, `rootRightSubtree`, and `rootLeftSubtree` as dictated by the interface in Figure 6.2. This is shown in the bottom portion of Figure 6.3. The `initialize` and `newTree` operations required by the specification will be handled by constructors when we actually implement a tree class; we can't put constructors in a Java interface because interfaces are abstract by definition and cannot be instantiated. The Java code for a complete `SimpleTree280<I>` interface follows. Try to understand all of the parts of it and why it is written the way it is. Note how the class header contains the clause `extends Container280`; this means that the interface also contains the method signatures therein and that any class that implements `SimpleTree280<I>` must also implement the methods in the `Container280` interface. The `ContainerEmpty280Exception` is an exception class defined in `lib280` which distinguishes exceptions resulting from the data structure being empty from other exceptions. Other than being a different type, it behaves like a standard `RuntimeException` and doesn't do anything unusual.

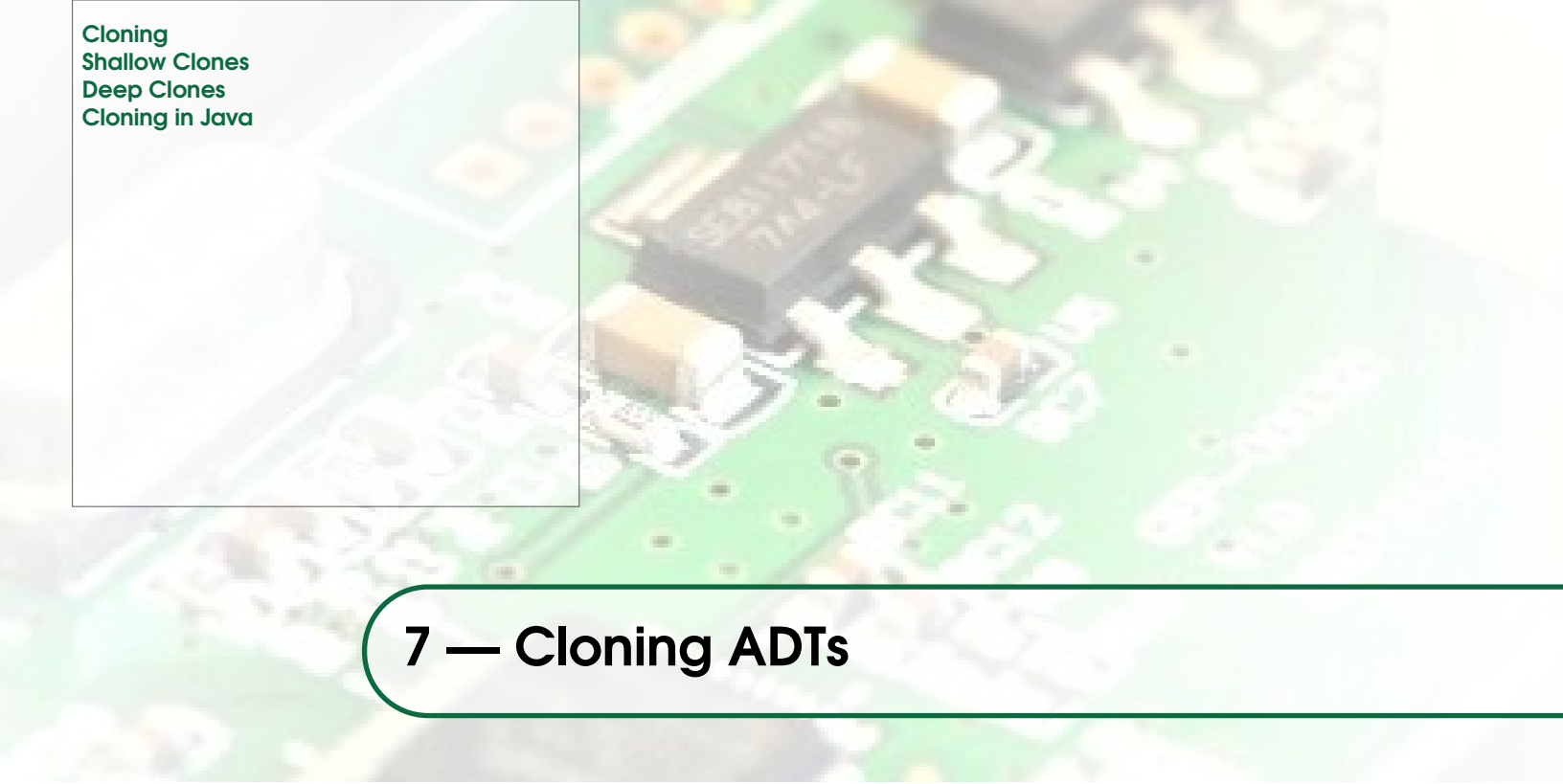
```
public interface SimpleTree280<I> extends Container280
{
    /**
      * Contents of the root item.
      * @precond !isEmpty()
      */
    public I rootItem() throws ContainerEmpty280Exception;

    /**
      * Right subtree of the root.
      * @precond !isEmpty()
      */
    public SimpleTree280<I> rootRightSubtree()
           throws ContainerEmpty280Exception;

    /**
      * Left subtree of the root.
      * @precond !isEmpty()
      */
    public SimpleTree280<I> rootLeftSubtree()
                  throws ContainerEmpty280Exception;
}
```

6.2.2 Implementing the `SimpleTree280<I>` Interface

Before we can proceed with writing a class that implements the `SimpleTree280<I>` interface, we need to have a class that stores information about nodes — we will call this `BinaryNode280<I>`. It will need to contain the following pieces of information:

- the data element (of type `I`) stored in the node;
- a reference to the root node of the left subtree;
- a reference to the root node of the right subtree;

We are now ready to implement a `BinaryNode280<I>` class and a class that implements the `SimpleTree280<I>` interface. We will see how to do this through in-class exercises. If you'd like some extra practice, see if you can come up with a `BinaryNode280<I>` class on your own, right now. Then you'll be able to see how closely it matches what we do in class.



7 — Cloning ADTs

Learning Objectives

After studying this chapter, a student should be able to:

- define what it means to *clone* an object;
- accurately describe the difference between *shallow* and *deep* cloning.

7.1 Cloning

Sometimes we want to make a duplicate copy of an instance of an ADT (object) and the data that it contains. This is referred to as *cloning*. For objects which contain only primitive data, cloning an object is easy: create a new object instance of same type as the object to be cloned, and copy the instance variables of the original object into the new object. However, if the object to be cloned contains references to **other** objects, then the process of cloning becomes more involved because we have to ask the very important question: *should we clone the objects that are referenced by the object being cloned?* For example, if we are cloning a `LinkedSimpleTree280<I>`, should we also clone the `BinaryNode280<I>` object referenced by the `LinkedSimpleTree280<I>` object? Let's suppose we decide that the answer is "yes", we should clone the root node object as well. But now the `BinaryNode280<I>` object we are about to clone contains references to **two** other objects (the left and right child nodes). Should we clone **them**? This chapter discusses the two situations depending on whether or not we choose to clone objects referenced by an object being cloned.

7.2 Shallow Clones

If we choose not to clone any objects referenced by the object about to be cloned, this is called a *shallow clone*. When performing a shallow clone, only one new object instance is created, namely an instance of the type of the object instance being cloned. Then the instance variables of the object instance being cloned are copied into the new object instance. If the instance being cloned contains any references, we just copy those references; we do not look any deeper for other referenced objects that could be cloned (hence the name *shallow clone*).

By way of example, suppose we have a variable `s` of type `LinkedSimpleTree<Character>`. This variable references an object of type `LinkedSimpleTree<Character>`, which in turn references a `BinaryNode<Character>`, which in turn references other nodes. The shallow cloning of `s` is illustrated in Figure 7.1. Notice how only one new tree object, `t`, is created, and that it refers to the nodes of the original tree `s`. While sometimes this is fine, other times a shallow clone can have unexpected or unintended side effects. For example, if we modify the new tree through the object instance `t`, we are also modifying the tree referenced by `s`, perhaps without even realizing it. Likewise, if we modify the nodes of the original tree `s`, we also modify the nodes of the clone tree `t`.

7.3 Deep Clones

A deep clone of an object clones all of the other objects referenced either directly or indirectly by the object being cloned. Think of deep cloning as doing a recursive shallow clone. A deep clone of an object consists of a shallow clone of that object, followed by recursively deep-cloning every object referenced by the object just cloned.

To illustrate the difference, suppose we were again cloning a `LinkedSimpleTree<Character>` object. In a deep clone, not only would we get a copy of the `LinkedSimpleTree<Character>` object, but that new tree object would reference clones of all of the original tree's nodes as well! This is illustrated in 7.2. Notice how every object that is reachable by following references from the original tree `s` has been cloned. In the case of deep cloning we can safely modify the cloned tree `t` and be assured that nothing in the original tree `s` will be changed. The tradeoff is that because of all the additional objects potentially being duplicated, a deep clone is potentially a much more expensive operation (in terms of time and memory space) than a shallow clone.

7.4 Cloning in Java

We will discuss shallow and deep cloning in Java in class.

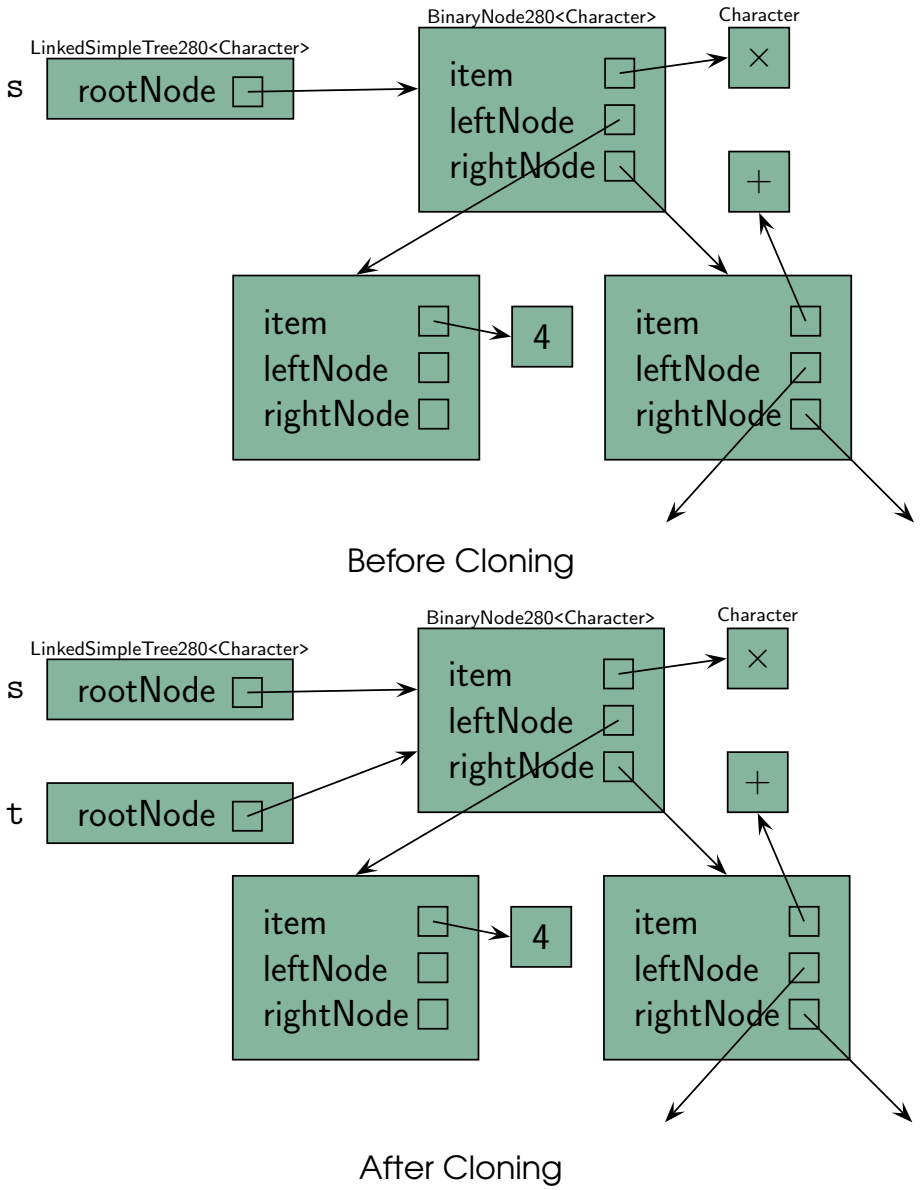


Figure 7.1: Shallow cloning of the `LinkedSimpleTree<Character>` object `s`. Top: before cloning; bottom: after cloning. Note that the only new object created is `t` which refers to a copy of the original `LinkedSimpleTree<Character>` object.

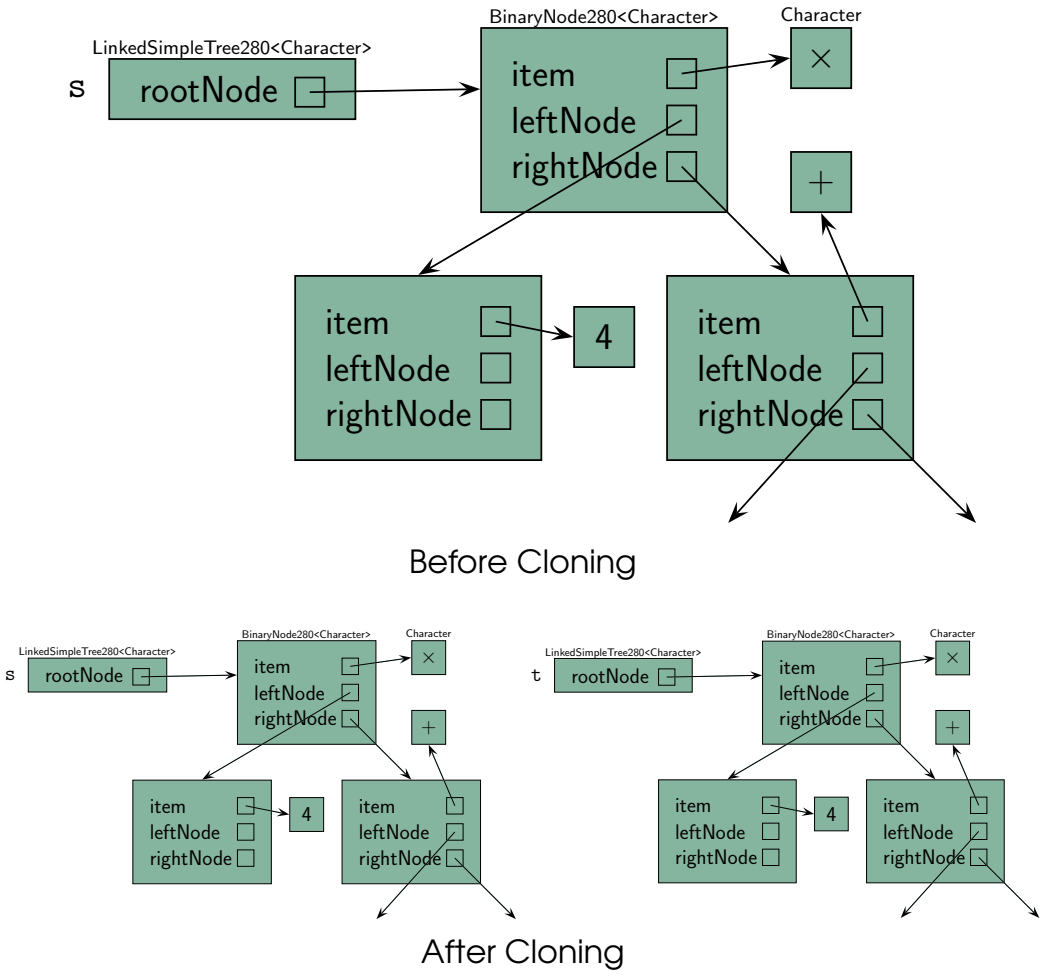


Figure 7.2: Deep cloning of the `LinkedSimpleTree<Character>` object `s`. Top: before cloning; bottom: after cloning. Note that **every** object referenced directly or indirectly by `s` was cloned and that `t` directly or indirectly references the cloned nodes.

8 — Tree Traversals

Learning Objectives

After studying this chapter, a student should be able to:

- describe the conditions under which nodes must be visited in a depth-first tree traversal;
- identify valid and invalid depth-first traversals of trees;
- describe the conditions under which nodes must be visited in a breadth-first tree traversal;
- identify valid and invalid breadth-first traversals of trees; and
- define and identify pre-order, post-order, and in-order traversals of binary trees.

8.1 Tree Traversals

As we know, it is very common to wish to perform some operation for all of the elements in a collection. For lists, this is easy because the data structure itself imposes a linear ordering on the elements, so we can just visit each element in the imposed order. Since the nodes (and hence, the collection of elements) in a tree do not have a well-defined linear ordering, the question arises: when we want to visit each node in a tree, in what order do we do so?

8.1.1 Traversals for General Trees

For general trees (in which each node may have any number of children), there are two main types of traversals: *breadth-first*, and *depth-first*. We can also consider another traversal, which is less common, much more difficult to implement, but sometimes desirable: the *level-order* search. We now consider each type of traversal in turn.

Depth-first Traversal

A depth-first traversal starts at the root, and always visits the children of a visited node before its siblings. Another way to say this is that once we visit a node t, we always visit every node in the subtree rooted at t before any other nodes not in the subtree of t. The following pseudocode outlines the algorithm:

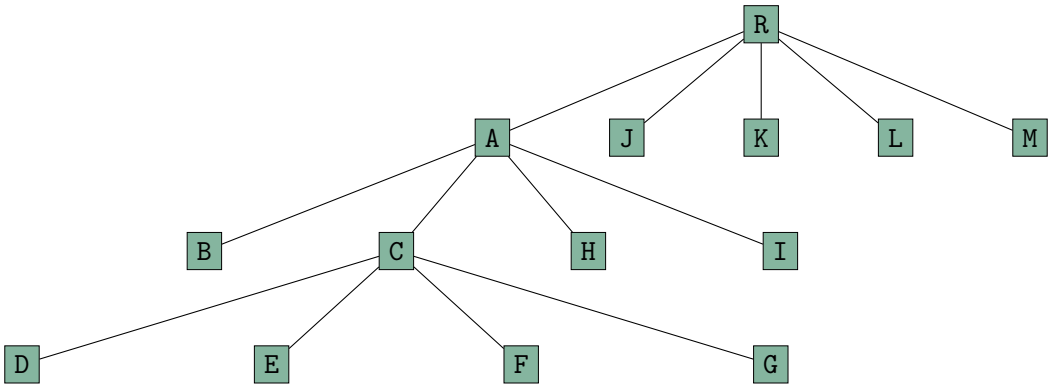


Figure 8.1: A meaningless tree for demonstrating traversal visit sequences.

```
Algorithm depthFirstTraversal(N)
Parameters:
    N is a tree node

visit node N
for each child T of N // Assume left to right, though it need not be so.
    depthFirstTraversal(T)
```

Now consider the tree in Figure 8.1. Suppose we call depthFirstTraversal with the node containing R as the parameter. You should now convince yourself that, if we follow the algorithm given above, and assume, as the comment in the algorithm suggests, that the for loop processes the children of `N` in order from left to right, then the nodes in the tree will be visited in the following order: R, A, B, C, D, E, F, G, H, I, J, K, L, M. Note, however, that the algorithm provides no guarantee that the loop works this way. We could, instead, assume that the children get processed by the loop from right to left, and get a different, but still valid depth-first traversal: R, M, L, K, J, A, I, H, C, G, F, E, D, B. We could also assume that the loop processes children in random order, and therefore R, J, L, K, A, C, D, E, F, G, H, B, I, M, is another valid depth-first traversal. The only requirement is that once we visit a node, we visit all its descendants before any other nodes.

Breadth-first Traversal

A breadth-first traversal starts at the root, and always visits all the nodes on the current level of the tree before visiting nodes on the next level; it does not, however, make any guarantees about the order in which nodes on a particular level are visited. Like the depth-first case, there may be many valid breadth-first traversals of a tree. Convince yourself that the following sequences of nodes represent valid breadth first traversals, remembering that the only requirements are that all nodes on level *i* get visited before any nodes on deeper levels get visited, and that we have to start at level 0.

- R, L, K, A, M, J, H, I, C, B, F, E, D, G
- R, M, L, K, J, A, I, H, B, C, D, E, G, F
- R, A, M, L, K, J, C, H, I, B, G, E, F, D
- R, A, J, K, L, M, B, C, H, I, D, E, F, G

The last sequence is a special case of breadth-first traversal called a *level-order* traversal. In a level-order traversal the nodes on each level get visited in order from left to right.

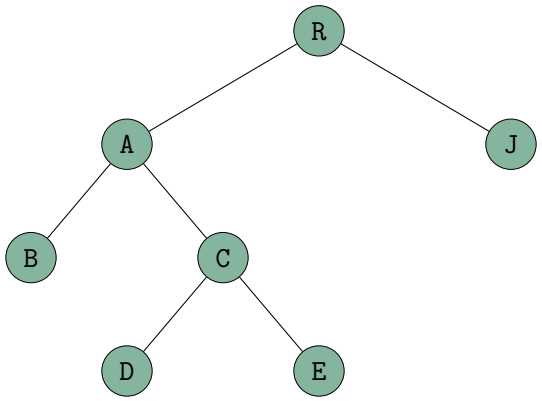


Figure 8.2: A meaningless binary tree for demonstrating pre-order, in-order, and post-order traversal visit sequences.

8.1.2 Traversals for Binary Trees

For Binary trees there are three specific traversals that have important significance. These are the *pre-order*, *in-order*, and *post-order* traversals. All of these traversals involve three operations:

V: visit the node
L: traverse the left subtree of the node
R: traverse the right subtree of the node

Pre-order Traversal

The pre-order traversal is a special case of the depth-first traversal. In the pre-order traversal, we start at the root, and then perform the above operations in the order: VLR. That is, every node is visited before both of its subtrees, and then we always traverse the left subtree before the right subtree. The pre-order traversal of the tree in 8.2 is: R, A, B, C, D, E, J. Note that it is a depth-first traversal since each node's descendants are always visited before any of its siblings. There is only one valid pre-order traversal of a binary tree.

In-order Traversal

The in-order traversal performs the traversal operations in the sequence: LVR, starting at the root. That is, the left subtree of a node is always traversed before the node is visited, and the right subtree of a node is always traversed after the node is visited. The in-order traversal of the tree in 8.2 is: B, A, D, C, E, R, J. There is only one valid in-order traversal of a binary tree.

Note that the in-order traversal is not a breadth-first traversal: left child of a node t is always visited before t, and t is one level above its children. The in-order traversal is also not a depth-first traversal because in a depth-first traversal a node t is always visited before any of its children. In an in-order traversal, a node t is always visited **after** its left child.

Post-order Traversal

The post-order traversal starts at the root and performs the traversal operations in the sequence: LRV. That is, both the left and right subtrees of a node t are traversed before t is visited. Again, this is neither a breadth-first, nor depth-first traversal. The post-order traversal of the tree in 8.2 is: B, D, E, C, A, J, R. There is only one valid post-order traversal of a binary tree.

8.2 Visits

Just knowing the order in which to visit nodes of a tree doesn't do much good unless we know what to do when each node is visited. Think of a "visit" as a placeholder for something that is done to the node, or something that is done with the data element in the node. Sometimes, it could be something done for the subtree rooted at the node. If we combine the right "visit" operation with the right kind of traversal, we can do some fairly interesting things. The following tasks can all be accomplished using traversals with a carefully chosen visit operation:

- Print the contents of all the nodes in a tree.
- Compute the height of a tree.
- Count the number of nodes in a tree.

We'll take a closer look at how to do these kinds of things in class.

9 — Ordered Binary Trees

Learning Objectives

After studying this chapter, a student should be able to:

- define what an ordered binary tree is and state the ordering properties that each node of an ordered binary tree must possess;
- explain the algorithm for determining whether an element exists in a binary search tree;
- explain the algorithm for inserting an element in an ordered binary tree; and
- list the operations that a full-featured binary search tree should support.

9.1 Ordered Binary Trees

A binary tree which just stores a collection of elements with no particular arrangement (other than the hierarchy imposed by the binary tree) isn't very useful. Consider the simple question: "Does a binary tree contain a certain element?" In an arbitrary binary tree, the only way we could answer that question would be to examine every node in the tree until we either find the element, or exhaust all of the nodes and conclude that the element is not present. This offers no advantage over a list, so why would we use a binary tree?

The answer, of course, is that if we impose some additional structure on a binary tree that takes advantage of the ability to arrange elements hierarchically, we can vastly improve search times, and the performance of other operations.

There are *many* ways we could introduce this additional structure, but we'll start with one that you already know: the *binary search tree* (BST) or *ordered binary tree*. If you remember (a whole year ago!) from CMPT 115, we did not allow duplicate elements in binary search trees. In this course we use a slightly different definition that allows for duplicate elements.

Definition 9.1 An *ordered binary tree* or *binary search tree* is a binary tree for which the following *ordering properties* hold for every node t:

1. Elements stored in the left subtree of t are strictly less than the element stored at t.

2. Elements stored in the right subtree of t are greater than **or equal to** the element stored at t.

This structure potentially allows for more efficient retrieval of specific elements from the collection, compared to a list or arbitrary binary tree, because we can potentially find an item in the tree without examining every node.

9.1.1 Review: Searching an Ordered Binary Tree

Suppose we are given an ordered binary tree T and an element e and wish to determine whether T contains e. You may recall that if we examine the element at the root of T, let's call it r, and find that $r \neq e$, then the e must be in the left subtree of the root of T if $e < r$, and in the right subtree of the root of T otherwise. This process continues recursively on the left or right subtree or until we either find a node containing e, or reach a leaf node that does not contain e. In the latter case we conclude that T does not contain e.

In general, given a node n of a BST, and an element e, the ordering properties of the tree tell us whether e will appear in the left or right subtree of n, a property which we take advantage of to speed up searches. The algorithm follows:

```
Algorithm has( n, e )
n: root node of a binary search tree
E: an element of the type stored in T

if n == null
    // Either we reached a leaf node, or the tree was empty
    return false;

if e == n.item
    return true
else if e < n.item
    return has( n.leftNode, e )
else
    return has( n.rightNode, e);
```

Without doing the analysis, we remind the reader that searching for the existence of an element in an ordered binary tree requires $O(\log n)$ time **on average**, where n is the number of elements in the tree. This is an improvement over searching for an element in an unstructured binary tree or a list, which would be $O(n)$ in the **best case**. However, we must also remind the reader that searching for an element in an ordered binary tree is still $O(n)$ in the **worst case**. Recall that a balanced binary tree with n nodes has $O(\log n)$ levels. A *degenerate* tree occurs when all of the children are on one side (see right side of Figure 9.1). A degenerate tree with n nodes has $O(n)$ levels. The BST `has` algorithm, in the worst case, looks at one node per level of the tree. Thus, in a balanced tree with $O(\log n)$ levels, we need only look at $O(\log n)$ nodes. But for a degenerate tree with n levels, we must look at $O(n)$ nodes.

9.1.2 Review: Insertion into a binary search tree.

The algorithm for insertion of a new element into an existing binary search tree is very similar to the `has` algorithm because we need to search the tree to find the correct insertion point for the new element. We begin at the root node, examine the element at the root, and determine whether our new element belongs in the left or the right subtree. This process continues recursively until we find the node where the subtree in which our new element belongs is empty — this is the correct insertion point. The algorithm follows:

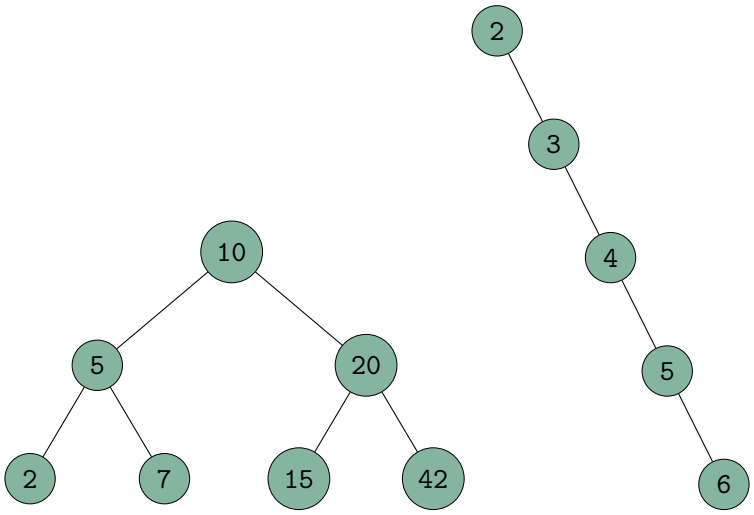


Figure 9.1: Left: a balanced BST with n nodes offers $O(\log n)$ search time because we only examine one node per level and there are $O(\log n)$ levels. Right: a degenerate tree, which has n levels; searching examines one node per level, making searching a degenerate tree $O(n)$.

```
Algorithm insert(n, e)
n: root node of a binary search tree
e: element to be inserted

if n == null
    // the tree was empty to start with
    make a new node containing e the root of the tree.
else
    if e < n.item
        if n.leftNode != null
            insert(n.leftNode, e)
        else
            s = new node containing e
            n.leftNode = s;
    else
        if n.rightNode != null
            insert(n.rightNode, e)
        else
            s = new node containing e
            n.rightNode = s;
```

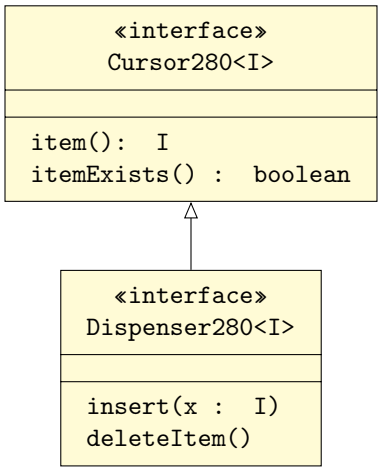
9.2 A Full-featured Ordered Binary Tree

Let's consider the operations that we would like to have in a full-featured ordered binary tree (binary search tree), and consider how we can fit what we want into `lib280`.

First and foremost, we want a binary tree, so it would make sense to extend the `LinkedSimpleTree280<I>` class.

Next, we want the ability to insert and delete elements from the collection. It turns out that

`lib280` has an interface called `Dispenser280<I>` which requires implementing classes to support `insert` and `deleteItem` operations. It looks like this:



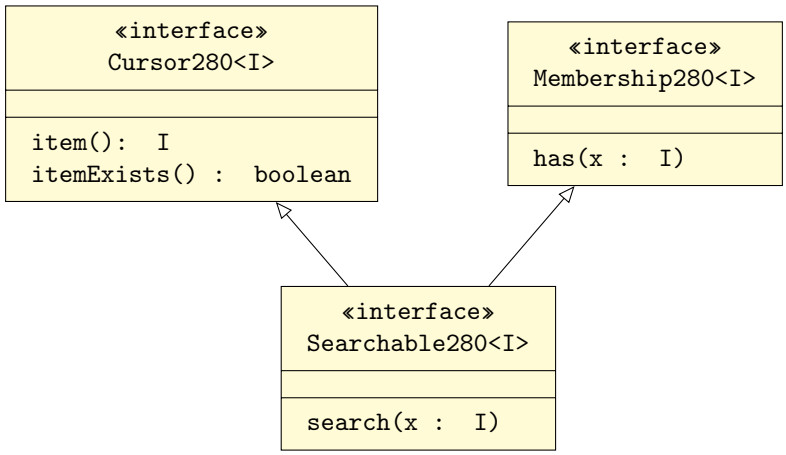
Ignoring for now why it is called "Dispenser," we merely observe that this interface supports two operations:

- `insert`: inserts an item into the container
- `deleteItem`: delete the current item

The `deleteItem` operation gives us a clue to why the `Dispenser280<I>` interface extends the `Cursor280<I>` interface — it says that the method `deleteItem` has to delete the *current item*. If there is to be a current item, then we have to know the position of the current item, and recording positions within a data structure is exactly what cursors do! So the `Dispenser280<I>` interface requires that any class that implements it implements the cursor methods `item` and `itemExists` as well. The "current item" is the item on which the cursor is positioned. A dispenser is a collection that allows us to do things to the current item, but doesn't allow other objects to manipulate the cursor position; in other words, our binary search tree will not have public cursor operations like `goFirst`, `goForth`, etc. It doesn't make sense to have them since the user doesn't get to decide **where** in the binary search tree a new item is inserted. The binary search tree determines that itself. To allow the user to move the cursor to a particular position and say "insert a new element at the cursor" doesn't make sense because it would then be quite possible to modify the state of the tree so that it is no longer an ordered binary search tree.

We have yet to discuss the algorithm for deleting an element from an ordered binary tree. This discussion will be delayed and treated as a separate topic because the algorithm, while not super-hard, is also not trivial and requires a good understanding of the other operations supported by ordered binary trees.

There are two other very important, and related operations that we would like to support. The first is `has` which simply returns a yes/no answer to the question "does the tree contain a given element?" Closely related is the operation `search` which is an operation that moves the cursor to a given element, if it is in the tree. In `lib280` there is an interface called `Membership280<I>` which specifies exactly the operation `has` that we described above. There is also an interface in `lib280` called `Searchable280<I>` which extends the `Membership280<I>` interface to include the `search` method. These are diagramed in UML below:



Again we see that `Searchable280<I>` also implements the cursor interface because the `search` method is a cursor manipulation method, so it requires that there be a cursor.

At this point, the reader may be wondering: why does `lib280` break all these operations up into such a large hierarchy of different interfaces, some of which only contain one method? The answer has a few different parts. Firstly, putting these common operations into interfaces helps to maintain consistency between ADTs in the library. If you want a container that is searchable, implement the `Searchable280<I>` interface, then all containers that are searchable have a method for searching that has the same name and signature. Secondly, grouping only the most closely related methods into small interfaces allows us to pick and choose more flexibly which operations a particular ADT should support, while maintaining the necessary dependencies — e.g. a searchable ADT has to have a cursor. Thirdly, interfaces like `Searchable280<I>` and `Membership280<I>` actually contain more than one method but for now we are ignoring the other methods that are not immediately relevant to the discussion.

We now know that we want our ordered binary tree to be, in `lib280` terms, a searchable dispenser that extends `LinkedSimpleTree280<I>`. We will take a closer look at the implementation in lectures. Figure 9.2 shows the complete UML diagram of all of the different classes and interfaces that will be inherited and implemented by `OrderedSimpleTree280<I>`.

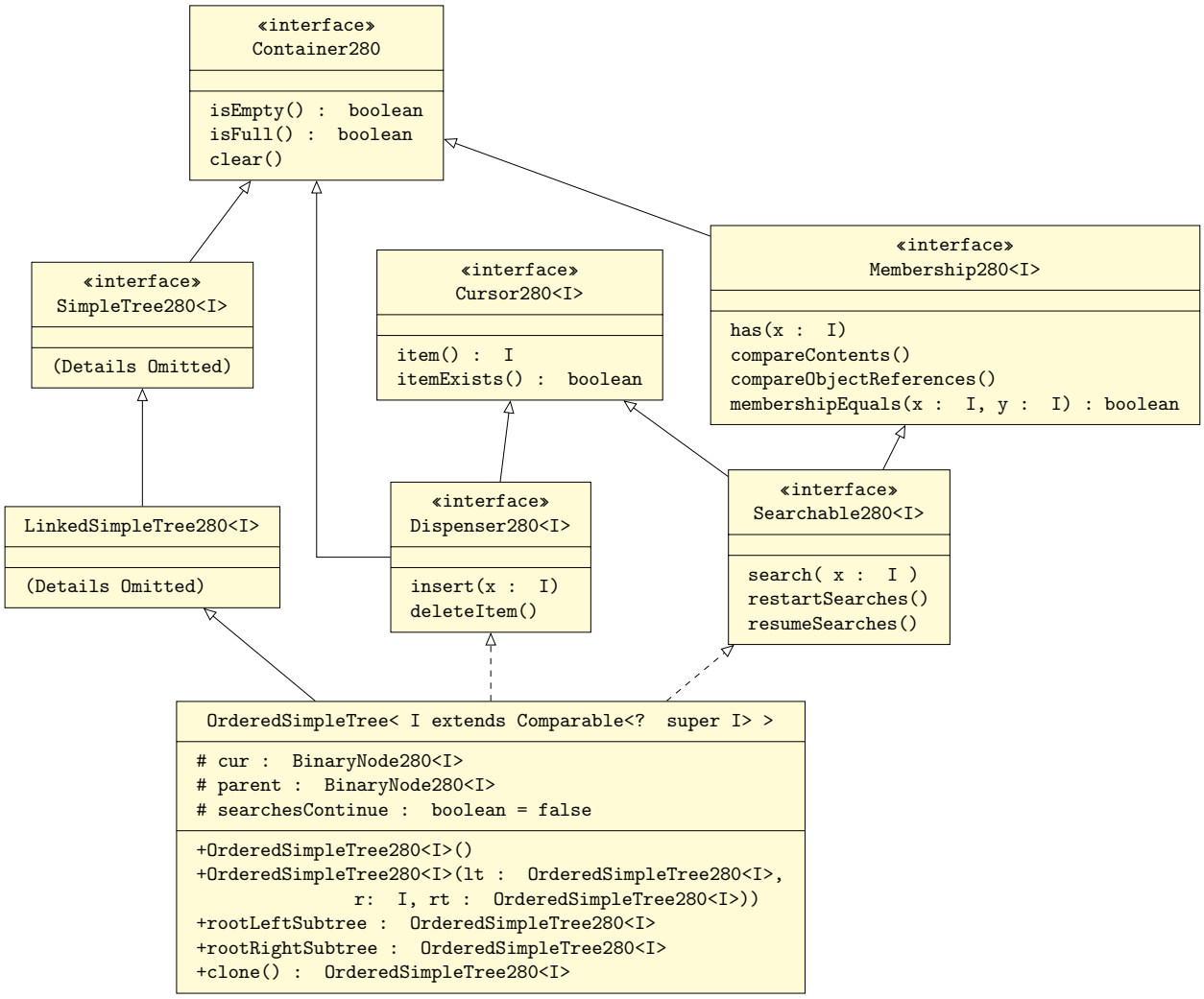
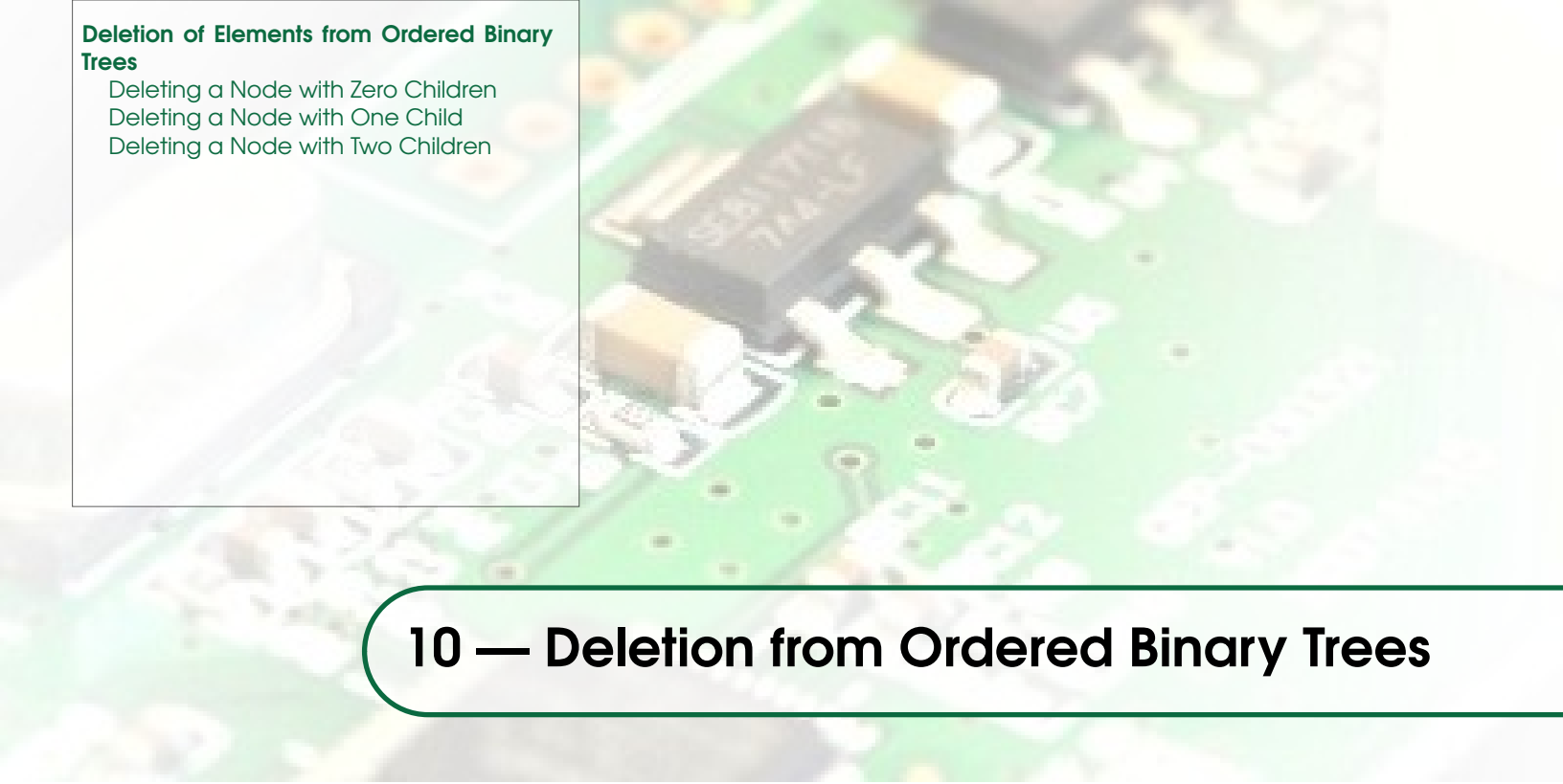


Figure 9.2: Class hierarchy for the OrderedSimpleTree280<I> class that implements an ordered binary tree (binary search tree) in `lib280`.



10 — Deletion from Ordered Binary Trees

Learning Objectives

After studying this chapter, a student should be able to:

- list the three possible cases that can arise when deleting an element from an ordered binary tree;
- define the terms *in-order successor* and *in-order predecessor*;
- explain the algorithms for deleting an element in each of the three cases; and
- given the drawing of an ordered binary tree, and a node to delete, draw the tree resulting from the deletion.

10.1 Deletion of Elements from Ordered Binary Trees

Insertion into an ordered binary tree is straightforward because new elements always replace empty subtrees. Deletion is more complicated because the element being removed might be in a node that has non-empty subtrees. The question then becomes: *what do we do with the subtrees of the deleted node?* There are three cases:

1. The node being deleted has zero children.
2. The node being deleted has one child.
3. The node being deleted has two children.

The first two cases are actually quite easy. It is the third case that requires the most effort to resolve. The solution, once you see it, is not so difficult to implement. The common theme among all three cases is that the node to be deleted, t, has a parent, p, and t is the root of either the left or right subtree of p. When we delete t, we must find a suitable replacement for the deleted subtree of p. The source of that replacement is different in each of the three cases of deletion. The key is to select a replacement for the deleted subtree that does not cause the ordering property of any tree nodes to be violated — that is, we need to choose a replacement such that the tree is still an ordered binary search tree.

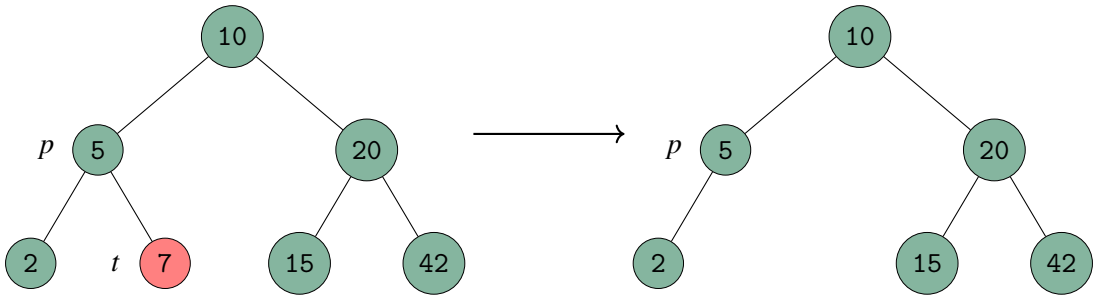


Figure 10.1: Deleting a node t that has zero children. The reference to t in node in p is replaced with null, the empty subtree.

In the remainder of this chapter, we will present the algorithms for the above three cases. In doing so, we will continue to refer to the node being deleted as t, and to t's parent as p.

10.1.1 Deleting a Node with Zero Children

This is the easiest case. If t has no children, then the requisite subtree of p can be replaced by an empty subtree without violating the ordering properties of any other nodes in the tree. The following pseudocode describes this case, and it is illustrated in Figure 10.1.

```
if t has no children
    let p be the parent of t
    replace the reference to t in p by null
    // Note: the reference to t in p might be either the left or
    // right child of p.
```

10.1.2 Deleting a Node with One Child

If t has only one child, r, then that child can become the replacement node for the subtree of p without violating the ordering properties. The following pseudocode describes this case, and it is illustrated in Figure 10.2.

```
if t has one child
    let p be the parent of t
    let r be the child of t
    replace the reference to node t in p by a reference to r
    // Note: the reference to t in p might be either the left or
    // right child of p.
```

10.1.3 Deleting a Node with Two Children

If t has two children, then it would appear that we are in trouble. One of the children could be linked to t's parent while preserving the ordering property, but then what should happen to the other child of t?

There are two common approaches for dealing with this problem. The one we will use is called *deletion by copying*.[1] Deletion by copying works by converting the deletion problem from one of deleting a node with two children to one of deleting a node with no more than one child.

[1] If you want to look it up, the other method is called *deletion by merging*.

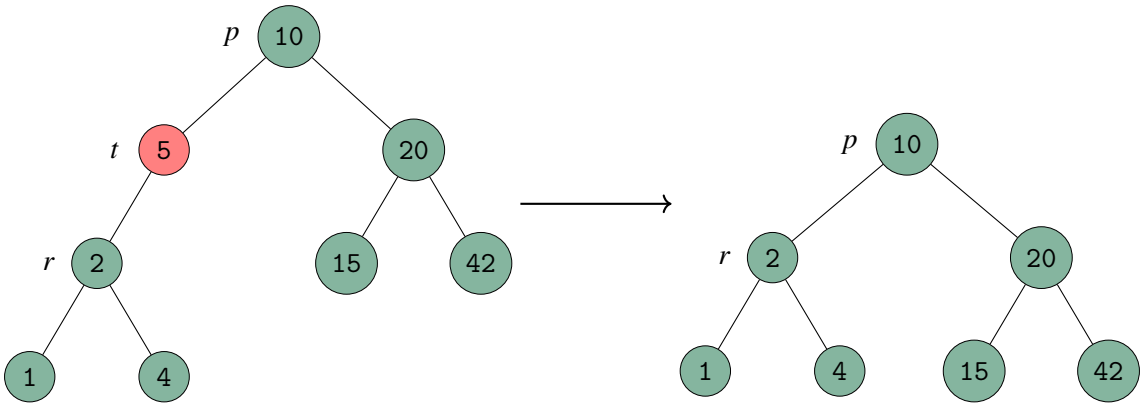


Figure 10.2: Deleting a node t that has one child. The reference to node t in p is replaced with a reference to node r (the child of t).

In-order Successor, In-order Predecessor

Given a node t, the *in-order successor* of t is the node containing the smallest element in the right subtree of t. We will need to find the in-order successor of t in the deletion algorithm for a node with two children. The in-order successor of t is so named because it is the node that would be visited immediately following the node t during an in-order traversal of the tree.

Similarly, the *in-order predecessor* of t is the node containing the largest element in t's left subtree. This is because it is the node that would be visited immediately prior to t during an in-order traversal of the tree. The in-order predecessor can be used in an alternate version of the *deletion by copying* algorithm.

Algorithm Details

Suppose we are deleting node t and that its left and right children are r, and s, respectively. Now, suppose we find the node that is the in-order successor of t; let's call the element stored there e. Further suppose that we copy e into node t, overwriting the element to be deleted, but not actually removing node t from the tree. Will this maintain the ordering properties? By definition, everything in the subtree rooted at s is larger than anything in the subtree rooted at r. So copying e to t will not violate the property that everything in t's left subtree have to be smaller than e. Moreover, since e was the smallest element in t's right subtree, then everything in t's right subtree must be greater than or equal to e. So problem solved! Well, except for the fact that we now have to delete the in-order successor of t because it contains the original (and now an extra) copy of e. How hard will it be to delete the in-order successor? Since the in-order successor contains the smallest element in the right subtree of t, the in-order successor cannot have a left child (otherwise there would have been a smaller element in the right subtree of t!). Therefore the in-order successor has at most one child, and is easy to delete. The algorithm is in the following listing, and an example is shown in Figure 10.3.

```
if t has two children
    find the element e in the in-order successor of t
    // i.e. the smallest element e in the right subtree of t

    copy e into t
    delete the in-order successor of t
```

Alternative Algorithm

Note that we could alternatively use the in-order predecessor instead of the in-order successor as a replacement for the element in t. The effect on the collection of elements is the same, but the ordered binary search tree itself will have a different structure.

```
// Alternate algorithm:
if t has two children
    find the element e in the in-order predecessor of t
    // i.e. the largest element e in the left subtree of t

    copy e into t
    delete the in-order predecessor of t
```

Some implementations make use of both options. For a given deletion, they choose the option which is most likely to maximize tree balance.

The Complete Deletion Algorithm

In the following listing, the algorithms for the three cases are combined to produce the complete ordered binary tree deletion algorithm.

```
if t has no children
    let p be the parent of t
    replace the reference to t in p by null
else if t has one child
    let p be the parent of t
    let r be the child of t
    replace the reference to node t in p by a reference to r
else
    find the element e in the in-order successor of t
    copy e into t
    delete the in-order successor of t
```

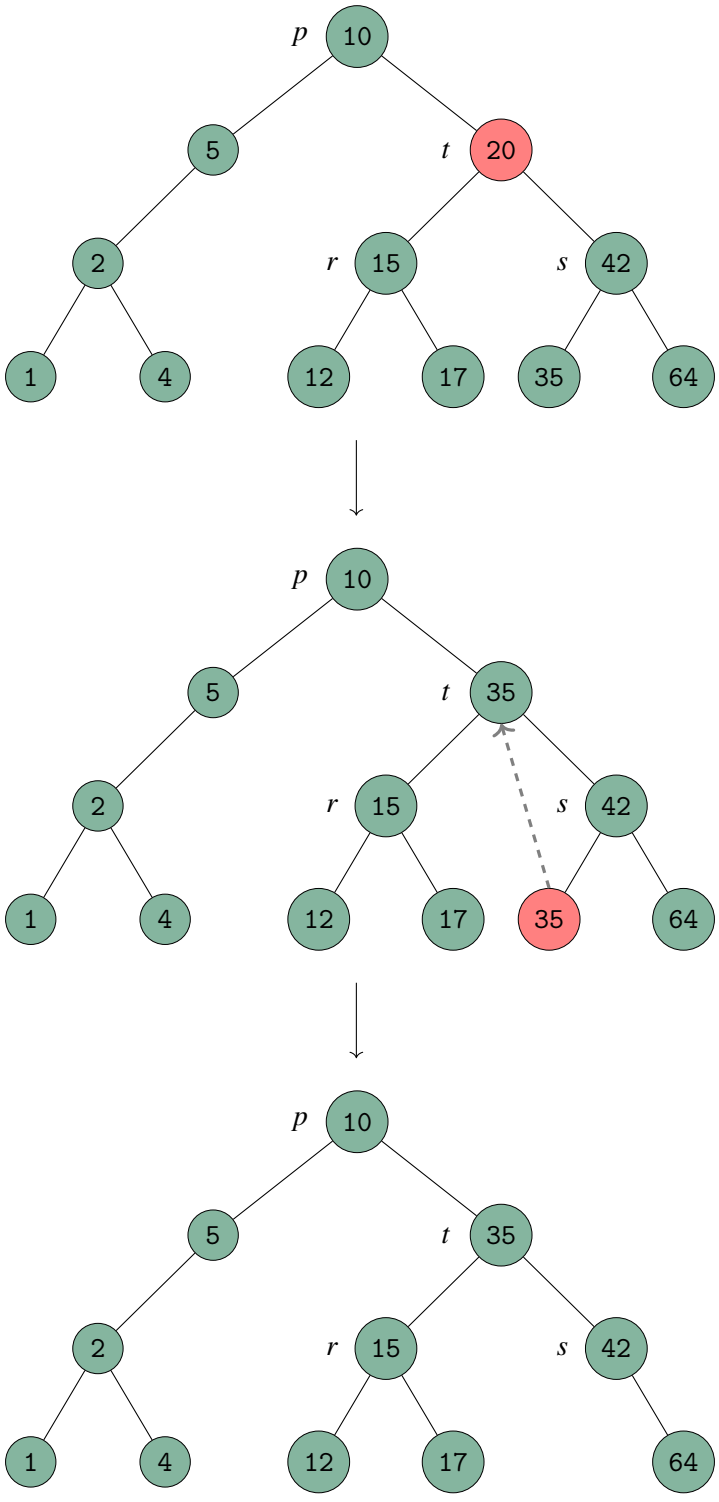


Figure 10.3: Deleting a node t with two children. Top: Before deletion of the element 20; middle: deletion in progress — the element in the in-order successor of t, 35, is copied into t; bottom: the in-order successor of t is deleted trivially and the removal of the element 20 is complete.

11 — AVL Trees

Learning Objectives

After studying this chapter, a student should be able to:

- explain what the *imbalance* of a tree node is;
- explain what the *maximum imbalance* of a tree is;
- explain the AVL tree property; and
- recognize the four situations of imbalance (which violate the AVL property) which can arise from insertion of an element into an AVL tree.

11.1 Balanced Trees

We have previously observed that searching in an ordered binary tree requires one comparison per tree level in the worst case. Furthermore, we have observed that in the worst case, a tree could be degenerate and have the same number of levels as nodes, like the tree in Figure 11.1. Thus we concluded that searching for an element in a degenerate ordered binary tree is $O(n)$ in the worst case.

A *perfectly balanced* binary tree is one in which there are n levels and every level is *complete*, that is, there are as many nodes as possible on each level. In such a tree there are exactly $\log(n+1)$ levels, which means we can conclude that searching in a perfectly balanced tree is $O(\log n)$. An example of a perfectly balanced ordered binary tree is given in Figure 11.2; note that it contains exactly the same elements as the degenerate tree in Figure 11.1.

From this we might conclude that if we could find some way to keep an ordered tree perfectly balanced, we could reduce the worst-case time complexity for searching to $O(\log n)$. The bad news is that we can't keep a tree perfectly balanced simply because perfectly balanced binary trees have exactly $n = 2^i - 1$ nodes for some $i \geq 0$. By definition, you can't have a perfectly balanced tree with 6 nodes because there would have to be some level that doesn't have as many nodes as possible. But you could easily have an ordered binary tree with 6 nodes, so what are we to do?

This problem can be solved by just... relaxing. We don't need a **perfectly** balanced tree to guarantee $O(\log n)$ searches, we just need an **almost** perfectly balanced tree. But this means we need a way to measure just how balanced or unbalanced a binary tree is.

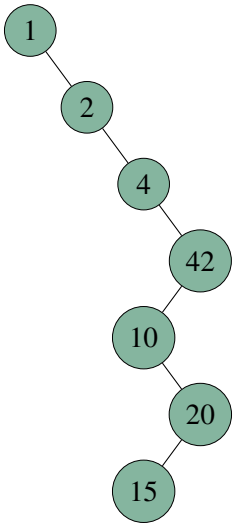


Figure 11.1: A degenerate ordered binary tree.

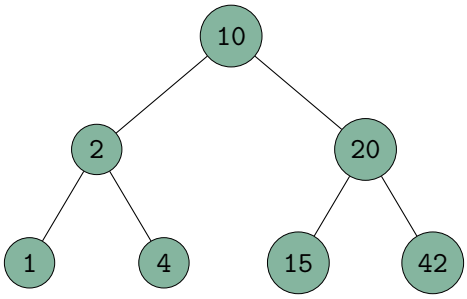


Figure 11.2: A perfectly balanced binary tree.

11.1.1 Imbalance and Maximum Imbalance

Definition 11.1 The *imbalance* of a node of a binary tree is the difference between the height of its two subtrees.

The *maximum imbalance* of a tree is the largest imbalance of all of the nodes in the tree.

It is important to note that *imbalance* is a property of a **node** in the tree, while *maximum imbalance* is a property of an **entire binary tree**. It turns out that it is quite possible to modify the insertion and deletion algorithms of an ordered binary tree such that we can guarantee that the maximum imbalance of the tree is at most 1. It also turns out that a guaranteed maximum imbalance of 1 is enough to guarantee $O(\log n)$ searches. A classic way of doing this is with AVL trees.

11.2 AVL Trees

The AVL tree is a classic example of one of many so-called *height-balanced* or *self-balancing* search trees that have been devised over the years. It is named for its creators, G. M. Adelson-Velskii and E. M. Landis, who published it in a 1962 paper.

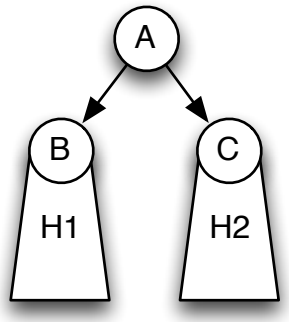
Definition 11.2 An *AVL tree* is an ordered binary tree such that, **every node** in the tree has the property:

$$|\text{height of left subtree} - \text{height of right subtree}| \leq 1.$$

In other words, an AVL tree is an ordered binary tree where every node has an imbalance of at most 1. In still other words, an AVL tree is an ordered binary tree where the maximum imbalance is no greater than 1. The trick is to figure out a way to make the insertion and deletion algorithms preserve this property, preferably in such a way that the insertion and deletion operations remain $O(\log n)$ time. If we can do this we will definitely succeed in guaranteeing an $O(\log n)$ search time because an AVL tree with n nodes has height $\leq 2\log n$ (proof omitted).

We can see that there is hope that we can restore the AVL property after an insertion or deletion from an AVL tree if we make the following observation: *insertion or deletion in an ordered binary tree will increase or decrease the height of any given subtree by at most 1.* Consider the following more concrete example:

Suppose we have an AVL tree with a root node A that has child nodes B and C which are roots of subtrees of heights H_1 and H_2 respectively:



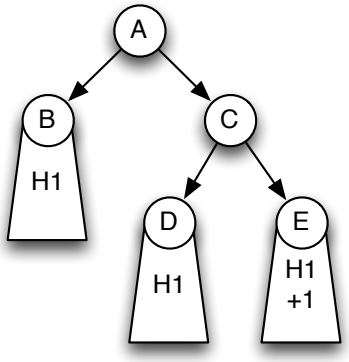


Figure 11.3: An RR imbalance. A is a critical node, because its imbalance is 2. The right subtree of A is heavier than its left, and the right subtree of C is heavier than its left, so we call this an RR imbalance.

Now let's suppose that we are inserting a new element e into this tree that happens to belong in the right subtree of node A, that is, the subtree rooted at C. Since this is an AVL tree, we know that H_1 and H_2 differ by at most 1. There are three cases to consider:

Case 1: $H_2 = H_1 - 1$ **before insertion.** In this case, the height of H_2 is one less than that of H_1, so if we insert into the subtree rooted at C, either its height doesn't change, or is increased by 1. That means that the height of the subtree rooted at C after insertion becomes at most $H_2 = H_1 - 1 + 1 = H1$, so the AVL property is maintained without any extra work.

Case 2: $H_2 = H_1$ **before insertion.** In this case the height of the subtree rooted at C again either doesn't change or is increased by 1, which means that, after insertion, $H_2 = H_1 + 1$ at most, and again, the AVL property is maintained without any extra work.

Case 3: $H_2 = H_1 + 1$ **before insertion.** In this case, still the height of the subtree rooted at C either doesn't change or increases by 1. If it does increase by 1, then we have a problem because after insertion we will have that $H2 = H1 + 2$, which means that A's imbalance is 2 and that violates the AVL property!

We can easily imagine the symmetric cases for when e is being inserted into the left subtree of A.

11.2.1 AVL Imbalance Cases

In general there are four situations that arise that require the AVL property to be restored. The first two arise from case 3 in the previous section. When $H2 = H1 + 1$ and e gets inserted into the right subtree of A, there are two cases depending on which subtree of C the element ends up in. In Figure 11.3 we see that the new element e ended up in the right subtree of C making C's right subtree "heavier" than its left. Also, A's subtree is heavier than its left subtree. Since both A and C are right-heavy, we will call this an RR imbalance. As long as the only node in the entire tree with an imbalance greater than 1 is A, we can fix this easily. A is called a *critical node* because it is the root of an imbalanced tree in which all subtrees are sufficiently well balanced. It is important to note that A is not necessarily the root of an entire tree. All of these situations also apply when A is the root of a subtree of some larger tree.

If, instead, e ends up in the right subtree of A but the left subtree of C then we have the situation where A is right-heavy, but C is left heavy, or an RL imbalance. Again A is a critical node — it has

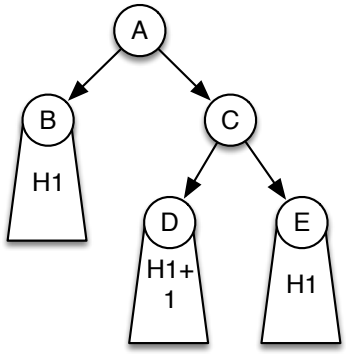


Figure 11.4: An RL imbalance. A is a critical node, because its imbalance is 2. The right subtree of A is heavier than its left, and the left subtree of C is heavier than its right, so we call this an RL imbalance.

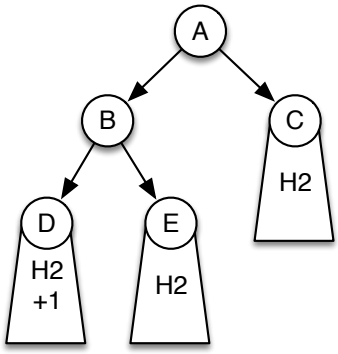


Figure 11.5: An LL imbalance. A is a critical node, because its imbalance is 2. The left subtree of A is heavier than its right, and the left subtree of B is heavier than its right, so we call this an LL imbalance.

an imbalance of 2 and its descendants have an imbalance of no more than 1. This is illustrated in Figure 11.4.

The other two situations that arise occur when the element e is smaller than the element stored in node A and the element ends up in a subtree of B, and, after insertion, $H_1 = H_2 + 2$. We could end up in the situation shown in Figure 11.5 where both A and B are left-heavy because e ended up in the left subtree of B. This is an LL imbalance, and A is a critical node.

When e ends up in the left subtree of A and the right subtree of B we get the situation depicted in Figure 11.6 in which A is left-heavy, and B is right-heavy. A is once again a critical node. Unsurprisingly, we call this an LR imbalance.

11.2.2 Repairing Imbalance

Having enumerated different ways in which an AVL tree might become imbalanced, we need to address the algorithms for repairing those imbalances. Repairing the AVL properties for the entire tree are accomplished using local adjustments to the tree called *rotations*. There is one kind of

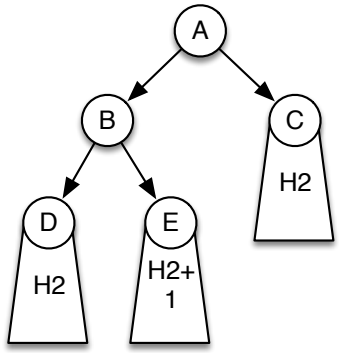


Figure 11.6: An LR imbalance. A is a critical node, because its imbalance is 2. The left subtree of *A* is heavier than its right, and the right subtree of *B* is heavier than its left, so we call this an LR imbalance.

rotation for each of the four imbalance situations that were presented in the previous section. We will examine these in detail in class.

12 — 2-3 Trees

Learning Objectives

After studying this chapter, a student should be able to:

- explain the properties of a 2-3 tree;
- explain the algorithm for searching for an element in a 2-3 tree;
- explain the process of splitting a 3-node;
- explain the algorithm for inserting an element into a 2-3 tree; and
- given a tree, be able to produce the result of an insertion on that tree.

12.1 Prelude: Key-item Pairs

Up to this point we have mostly illustrated container ADTs in which the elements are of a primitive data type. Suppose we want to store elements which are of a compound data type, but we want to keep them in some sort of order (e.g. in an ordered tree!). In order to establish an ordering on a compound data type, each instance of such a compound data type must have a unique *key* which can be easily compared to determine which of two elements is "larger." A key could simply be one of the pieces of primitive data that make up the compound data type, or it could be a function of one or more of the pieces of primitive data that make up the compound data type. In either case, the compound items in the container can be ordered by their keys.

As an example, let's suppose that we are building a database that contains information about famous pirates and that we have created a compound data type that stores, for each pirate,

- their name;
- the number of ships in their fleet; and
- the reward for their capture.

Since we'd most likely be looking up pirates in such a database by name, we should make the key for this data type the `name` field. We could then store several elements of this type in an ordered tree, which would keep the pirates in "order" by their name. Then, we could search the tree for a particular pirate, say "Hobblin' John Hawkins." If found, we could ask for the entire item with that

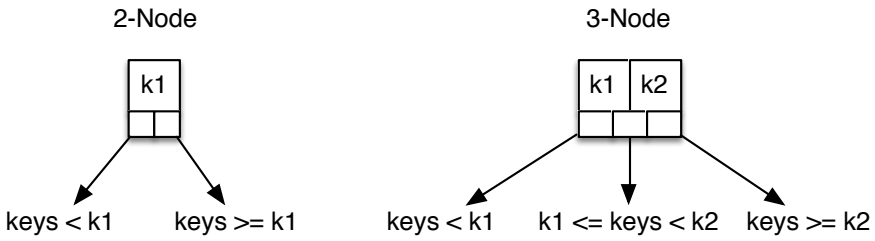


Figure 12.1: Left: a 2-3 node with two children; right: a 2-3 node with three children.

key which would tell us how many ships good ol' Captain Hawkins commands, and how much richer we would be if we brought him to justice. In short, the idea is to use a single key, which is usually (but not always) a primitive data type, to look up the entire data item that has a matching key.

Key-item pairs can be used with AVL trees or plain old ordered binary trees as well as 2-3 trees, we just didn't introduce the notion until now. In addition, this is the whole point of requiring that our items implement the `Comparable` interface. The implementation of `Comparable` effectively determines an item's key based on which data is used by the `compareTo` method to perform the comparison.

12.2 2-3 Trees

A 2-3 tree is another example of a *height-balanced* or *self-balancing* tree. A 2-3 tree is an ordered tree in which every node has exactly two or exactly three children. All leaf nodes in a 2-3 tree are on the same level. Each leaf node stores a key-item pair. Internal nodes do not store items.

Internal nodes store either one or two *key* values. If an internal node has two children (left and right), then it has only one key value k_1. The items found in the left subtree all have keys that are strictly smaller than k_1. Any items in the right subtree must have keys greater or equal to k_1. This kind of internal node is illustrated on the left side of Figure 12.1. If, however, an internal node has three children (left, middle, and right), it has **two** keys, k_1 and k_2. The left subtree contains items with keys strictly less than k_1. The middle subtree contains items with keys greater than or equal to k_1 and strictly less than k_2. The right subtree contains items with keys greater than or equal to k_2. This kind of internal node is illustrated on the right side of Figure 12.1. In class, we will see how both types of node can be implemented using a single class/object which stores everything needed for a 3-node, but also keeps track of which keys and child references are actually used by the node, depending on whether it is representing a 2-node or a 3-node.

12.2.1 Searching 2-3 Trees

Searching a 2-3 tree for an element with a particular key proceeds very much like searching in an ordered binary tree. The key being searched for is compared to the keys in the current node, and it is determined which subtree of the current node must contain the element with that key (if it exists). Then the search proceeds, recursively, in that subtree. A major difference between searching a 2-3 tree and searching an ordered binary tree is that since only the leaf nodes of a 2-3 tree contain key-item pairs, and since the leaf nodes are all on the same level, the search will always proceed to the lowest level of the tree.

In Figure 12.2 we see an example of a 2-3 tree where the keys are integers (for brevity, only the keys are shown in the leaf nodes, the corresponding items are omitted — or perhaps this is a 2-3 tree which just stores single integers). The green arrows show the sequence of subtrees considered if we search for the element with key 40. Since the root node only contains one key, 6, we proceed right because our key is larger than 6. On the second level of the tree, the key is larger than $k_2 = 30$, so we know that the element with key 40 must, again, be in the right subtree. At the third level of the tree, our key is equal to k_1 and strictly smaller than k_2, so the element with key 40 must be in the middle subtree. At the fourth level, we reach a leaf node. Since the key in the leaf node matches the key we are searching for, we have found the desired element. If, instead, we searched for the key 41, we would make all of the same decisions and still arrive at the node containing key 40 (verify this for yourself!). We would then have to conclude that there is no element in the tree with key 41. The algorithm for searching in a 2-3 tree follows.

```
Algorithm search(i, n):
i - key of element to be found
n - root node of tree in which to search

    if n is an interior node
        if i < k1
            search(i, n.left)
        else if there is no second key value or i < k2
            search(i, n.middle);
        else  // i >= k2
            search(i, n.right);
    else // i is leaf node
        if i = key of element in this node
            // found the desired leaf; possible actions we could take here:
            //     move cursor to n
            //     return the item in n
            return
        else
            // the key does not exist in the 2-3 tree; possible actions:
            //     invalidate cursor position
            //     return null
            return
```

12.2.2 Insertion of Key-element Pairs into a 2-3 Tree

Insertion is a recursive process somewhat similar to insertion into AVL trees. The basic ideas are the same: first, we recurse from the root to the leaf level (in the same way we do in a search) to determine where the new node should be inserted; second, as the recursion returns back up the tree, we perform local adjustments to restore any required properties of the 2-3 tree that were lost after insertion.

The insertion process begins by considering two special cases: an empty tree, and a tree with just one leaf node. If the tree is in neither of those states, we proceed with the recursive insertion algorithm by calling an auxiliary procedure. The pseudocode for handling these special cases follows.

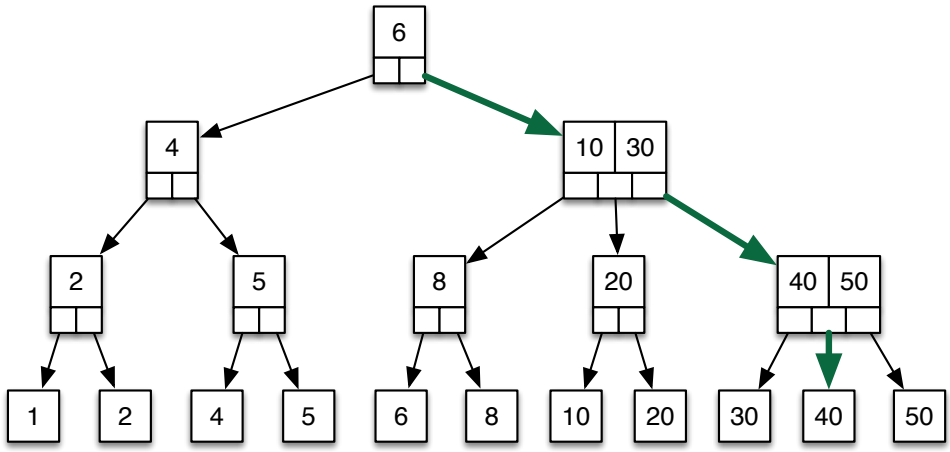


Figure 12.2: An example 2-3 tree. All key-element pairs are stored in leaf nodes. All leaf nodes are on the same level. For brevity we only show element keys in leaf nodes. The green arrows shows the sequence of subtrees traversed while searching for the element with key 40.

```
Algorithm insert(i, k)
The insertion operation for a 2-3 tree.

i - element to be inserted
k - key of element to be inserted

if the tree is empty, create a single root (leaf) node containing i
else if the tree contains a single leaf node m:
    create a new leaf node n containing i
    create a new internal node p with m and n as its children,
        and with appropriate key p.k1
else
    call auxiliary method insert(this.root, i, k)
```

Before we present the algorithm for the auxiliary procedure, let's look at an example. Suppose that we wanted to take the tree in Figure 12.2 and insert the element 15. First, we have to recurse down to the tree level just above the leaf level, and find out where the new leaf node should be inserted. This process is illustrated in the top portion of Figure 12.3. The green arrows show the branches that were taken in recursing down the tree. We can then see that 15 should become the middle child of the 2-node containing the key 20. This isn't a big problem. The 2-node containing the key 20 is converted to a 3-node and its keys are adjusted accordingly, as shown in the bottom portion of 12.3.

In the previous example, things could have been much worse. What would have happened if the leaf node for the new element had to become the child of a node that was already a 3-node? By now, you've likely noticed that the keys in the leaf nodes are ordered from left to right. It is, therefore, easy to see what would happen if we tried to insert an element with key 45. An element with key 45 would need to appear between the leaf nodes containing the elements with keys 40 and 50. Thus, it would have to become the child of the internal node with keys 40 and 50, but there is no room for such a child because we now have four child nodes and a parent that can't have more than three

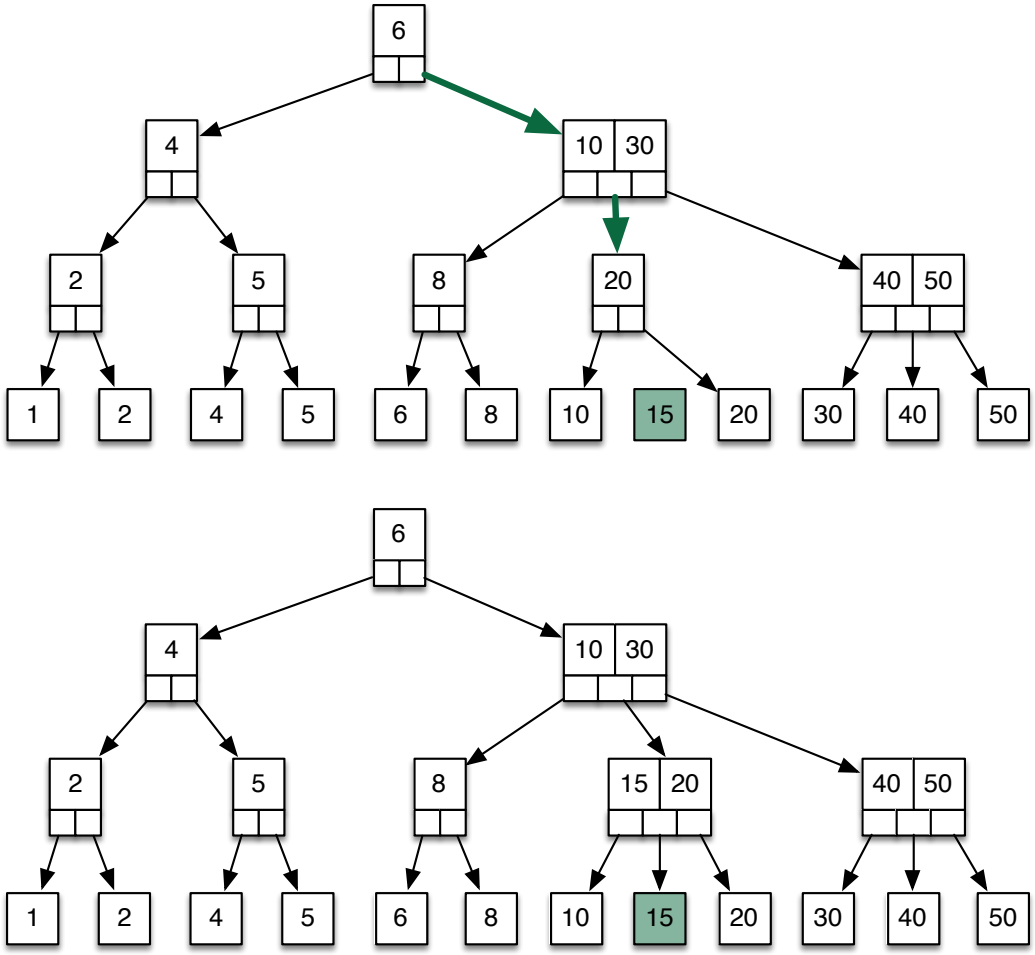


Figure 12.3: Inserting element with key 15 into a 2-3 tree. Top: the first step is to find where the element should be positioned at the leaf level. Green arrows show recursive calls down to the second last level. Bottom: the 2-node containing key 20 is converted to a 3-node to make room for the new element, and its keys are adjusted accordingly.

children. The solution to this problem is to split the 3-node, which needs to have four children, into two 2-nodes, and distribute the four children among them, two to each 2-node, in the proper order. The exact outcome of the split depends on where the new key appears in the ordering of the four leaf nodes being redistributed.

Consider Figure 12.4 which shows the complete sequence of events that occur when we insert the element 45 into the tree pictured in Figure 12.2. First, we recursively call insert in a search-like fashion and recurse down to the internal node with keys 40 and 50. We find that the element with key 45 has to be placed between the leaf nodes containing the elements with keys 40 and 50. This is shown in the top tree in Figure 12.2. However, the parent of the leaf nodes containing elements with keys 40 and 50 is a 3-node, so we have to split it by converting it into a 2-node with the smallest two of the four leaf keys as its children, and a new 2-node with the two leaf nodes that have the larger keys as children. The result of the split is the middle tree in Figure 12.4. Now the new 2-node with key 50 has to become a child of the red internal 3-node with keys 10 and 30. This causes another split; the 3-node with keys 10 and 30 is converted into a 2-node containing key 10 and a new 2-node containing the key 45. This new 2-node has to become a child of the root of the tree. Fortunately, the root of the tree is a 2-node, and can be converted to a 3-node to accommodate the new child. This is shown in the bottom tree of Figure 12.4.

Insertion: Node Splits

The splitting of nodes just above the leaf level is fairly straightforward, though there are four cases. If insertion requires that a new leaf node become a child of an existing 3-node p, then we have to consider four leaf nodes: the children of p, which we will call c_1, c_2, c_3, and c, the new leaf node. We sort these nodes in ascending order according to their keys, and the two smallest become children of p and the two largest become children of a new node q. Then `insert` returns a pair (q, k_s) where k_s is a key that can be used to partition between the keys that are children of p, and the keys that are children of q. In other words, for this base case, k_s is the third key in the sorted order of c_1, c_2, c_3, and c.

When a recursive call to `insert` returns (q, k_s), it means that q is a new node resulting from a split lower down the tree that has to be linked in as a child of the current node p, and that k_s is at least as small as any elements in the subtree rooted at q. There are five possible cases. The first two are easy to handle as these are the cases where the current node p is a 2-node, and so q can just be linked in at the appropriate position. This is illustrated in Figure 12.5. Note how, in both cases, k_s is used as a new key value in p.

If a recursive call to `insert` returns (q, k_s) and the current node p already has three children, then we have to split p. There are three cases depending on where q needs to be inserted, which are illustrated in Figure 12.6. After the split, the newly created node r is returned along with a key appropriate for partitioning between the keys in the subtree rooted at p and the subtree rooted at r.

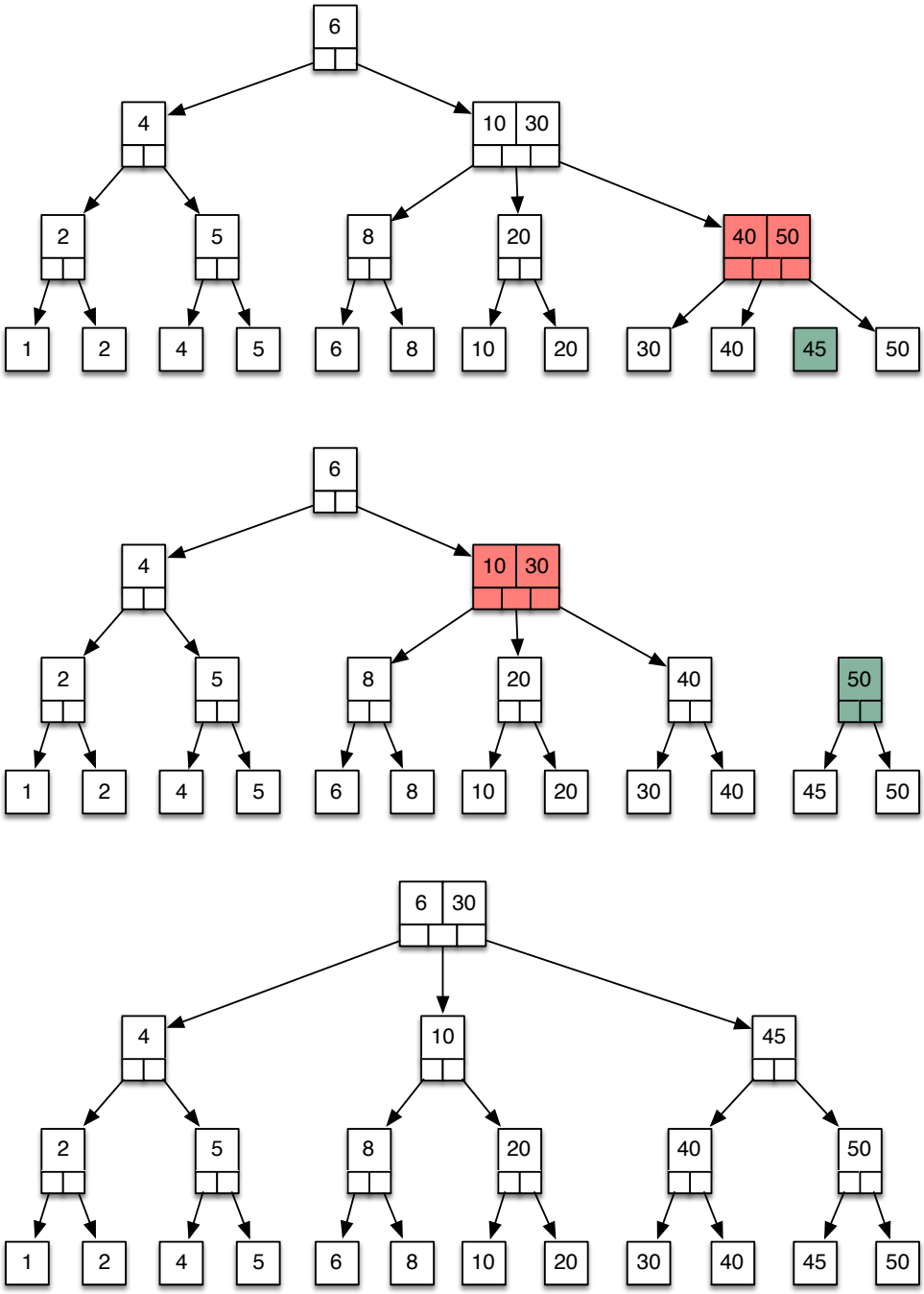


Figure 12.4: From top to bottom, these three trees show the sequence of events when the element with key 45 is inserted into the tree pictured in Figure 12.2. Green nodes represent new nodes (possibly caused by splits); red nodes indicate nodes about to be split to accommodate a green node. First the new leaf node containing key 45 is created (top). Then the 3-node containing 40/50 is split to accommodate the 45 (middle). This creates a new 2-node containing 50. The 3-node containing 10/30 is then split to accommodate the new 2-node containing 50 (bottom).

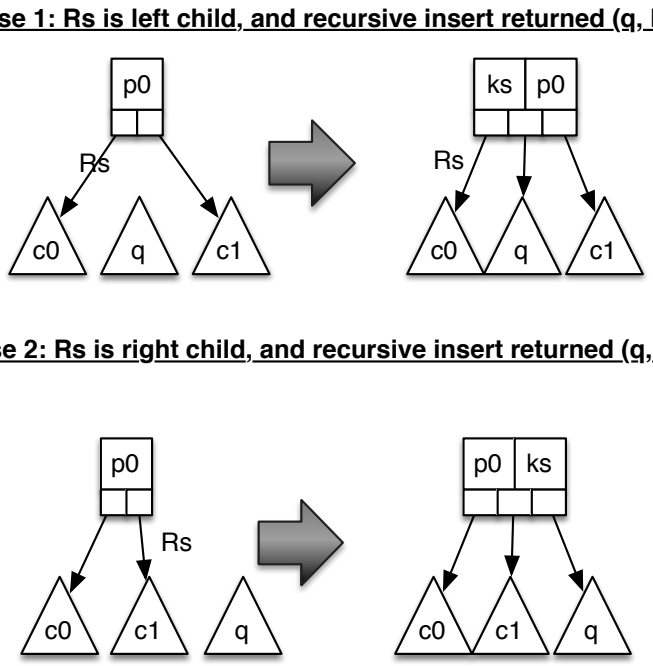


Figure 12.5: Linking a new node to a 2-node. R_s is the subtree that was followed when recursing down the tree and from which we have just returned. If (q, k_s) was just returned by that recursive call, and the current node p is a 2-node, node q is added as a child of p. k_s is used for the new key of p.

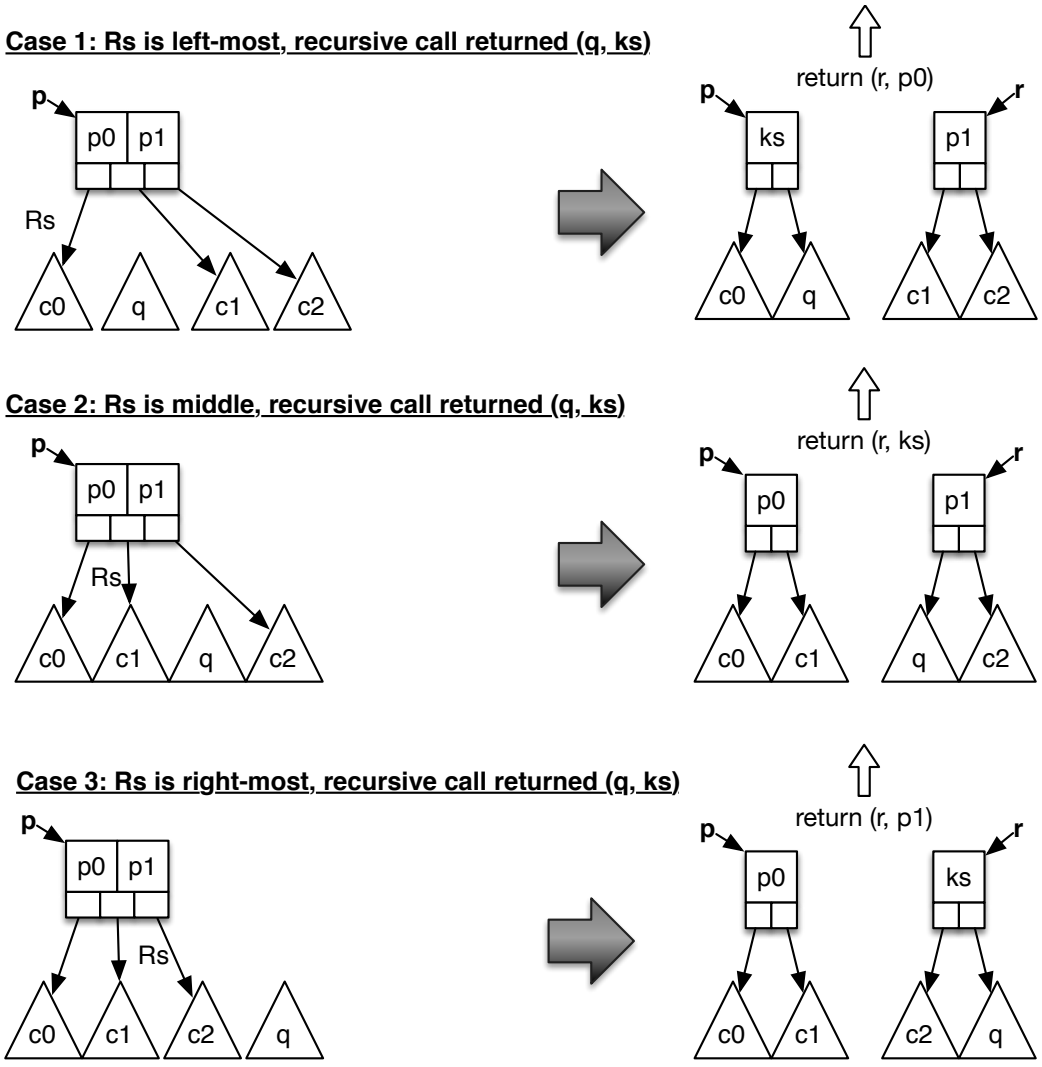


Figure 12.6: Linking a new node to a 3-node. R_s is the subtree that was followed when recursing down the tree and from which we have just returned. p is the current node to which q needs to be attached (q resulted from the split of a child of p). There are three cases depending on which subtree of p was traversed during the downward recursion. After the split, the new node r and a key appropriate for partitioning between the keys in p and the keys in r is returned.

Recursive Insertion Algorithm

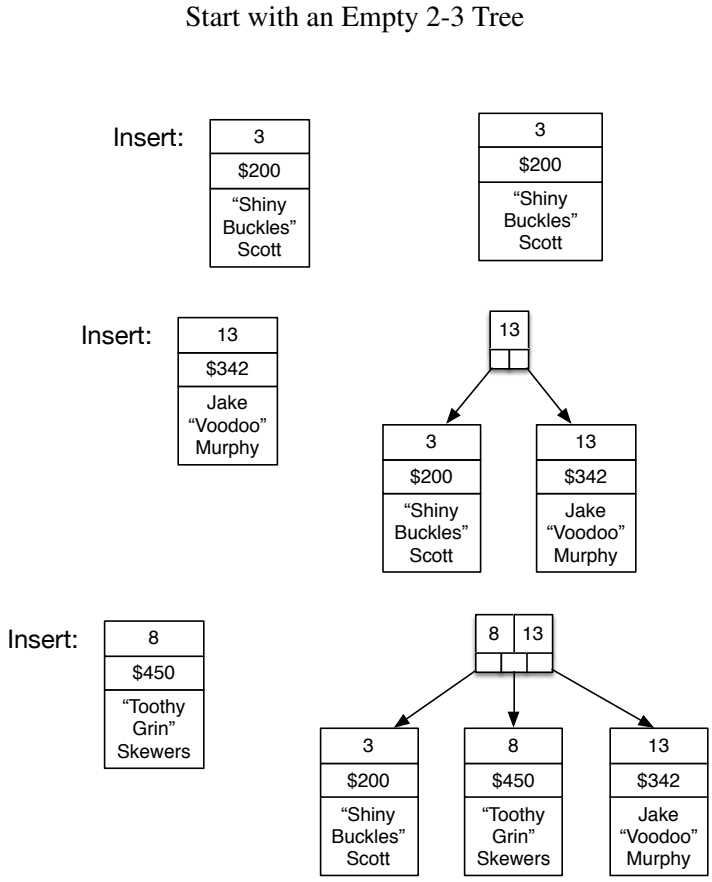
The complete insertion algorithm is shown below. Further discussion of the insertion algorithm shall take place in-class.

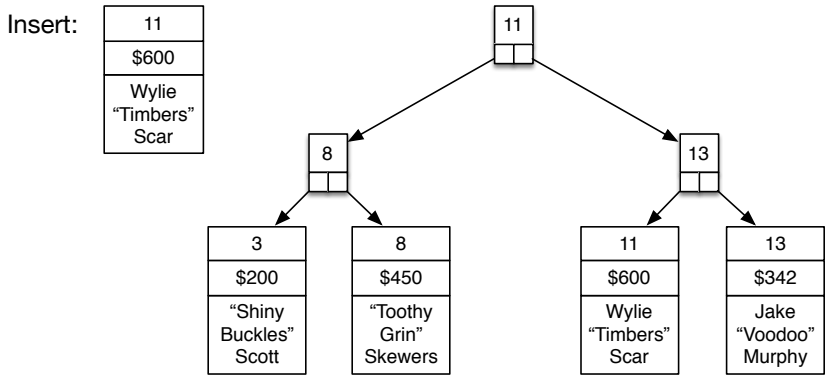
```
Algorithm insert(i, k)
The insertion operation for a 2-3 tree.

i - element to be inserted
k - key of element to be inserted

if the tree is empty, create a single root (leaf) node containing i
else if the tree contains a single leaf node m:
    create a new leaf node n containing i
    create a new internal node p with m and n as its children,
        and with appropriate keys p.k1 and p.k2
else
    call auxiliary method insert(this.root, i, k)


Algorithm insert(p,i,k):
This is the auxiliary recursive algorithm called by the
previous insertion algorithm, above.
p is the root of the tree into which to insert (k,i)
i is the element to be inserted
k is the key of the element i

if the children of p are leaf nodes  // base case
    create new leaf node c containing (k,i)
    if p has exactly two children
        make c the appropriate child of p, adjust p.k1, p.k2
        return null
    else // p already has 3 children
        let p1, p2, p3 be the three children of p
        sort keys of c, p1, p2, p3 in ascending order
        make smallest two keys the children of p
        make largest two keys the children of a new internal node q
        set keys in p and q according to the keys in their middle children
        ks = third largest key of {c, p1, p2, p3}
        return (q, ks)  // but now q needs a parent.
                        // attach q to the parent of p as the recursion unwinds
                        // by returning it.

else    // recursive cases
    if k < p.k1
        Rs = p.left;
    else if k < p.k2 or p has only 2 children
        Rs = p.middle
    else
        Rs = p.right

    (n,ks) = insert(Rs, i, k)
    if n is not null // n is new node resulting from a split, needs a parent.
                     // Make it the child of p
        if p has exactly two children
            // This will be one of the case illustrated in Figure 12.5
            make n the appropriate child of p.
```

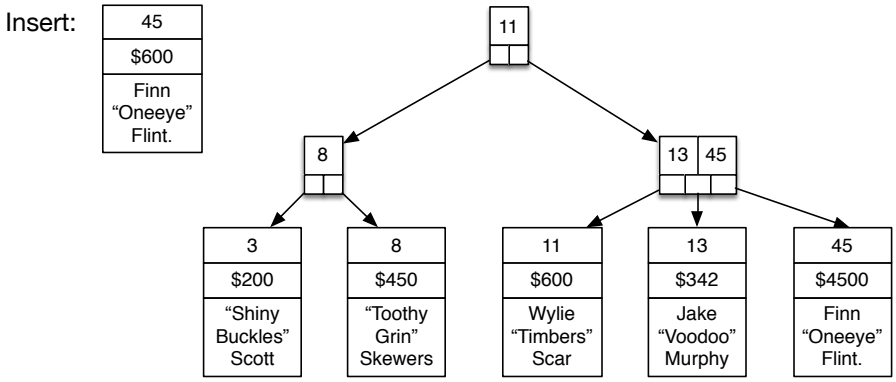
```
            update p.k1 and p.k2 appropriately using ks
            return null
        else // p already has 3 children, split p, return q
             // to attach to parent of p
             // This will be one of the cases illustrated in Figure 12.6
             // the split() function determine which case, and performs the
             // adjustments to the tree.
            (q, ks) = split(p, n, Rs, ks)
            return (q, ks)
```

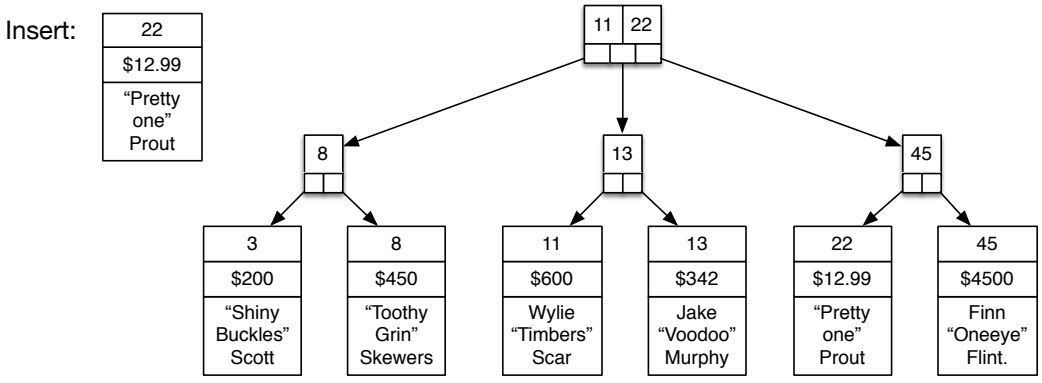
12.3 Additional Example

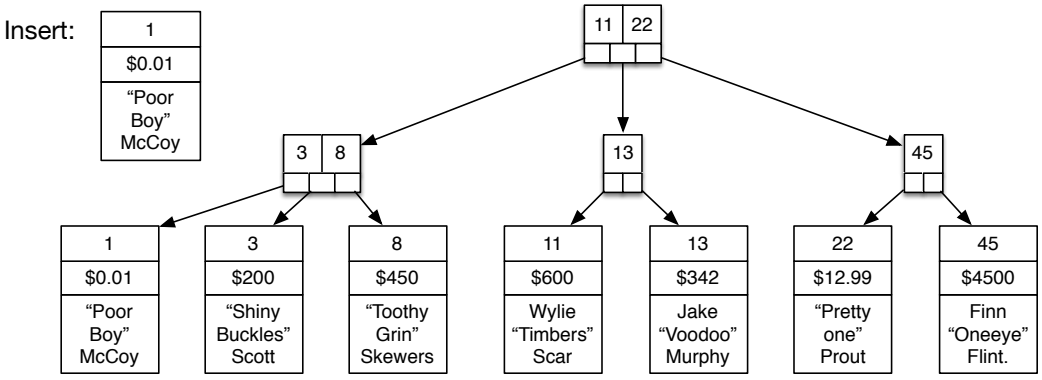
Here is an additional example of 2-3 tree insertion. For each insertion pictured, try to trace through the insertion algorithm, above, to see how it was arrived at. For this example, the elements are the data on pirates as in Section 12.1 (pirate name, number of ships in their fleet, and reward for their capture), only this time, we'll use the number of ships in their feet as the element's key.

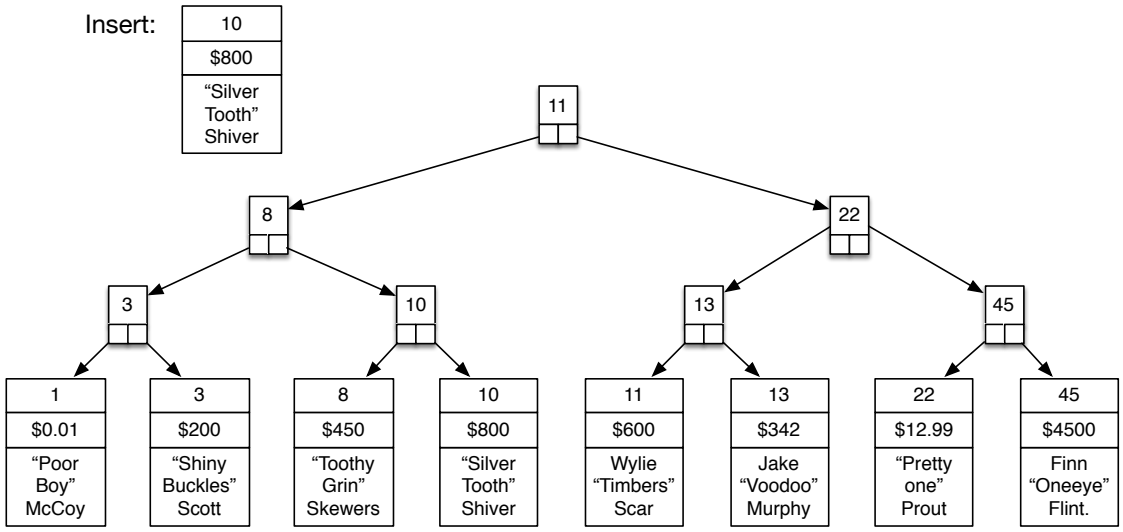



Insert:
11
$600
Wylie "Timbers" Scar
11
8
13
3
$200
"Shiny Buckles" Scott
8
$450
"Toothy Grin" Skewers
11
$600
Wylie "Timbers" Scar
13
$342
Jake "Voodoo" Murphy


Insert:
45
$600
Finn "Oneeye" Flint.
11
8
13 45
3
$200
"Shiny Buckles" Scott
8
$450
"Toothy Grin" Skewers
11
$600
Wylie "Timbers" Scar
13
$342
Jake "Voodoo" Murphy
45
$4500
Finn "Oneeye" Flint.


Insert:
22
$12.99
"Pretty one" Prout
11 22
8
13
45
3
$200
"Shiny Buckles" Scott
8
$450
"Toothy Grin" Skewers
11
$600
Wylie "Timbers" Scar
13
$342
Jake "Voodoo" Murphy
22
$12.99
"Pretty one" Prout
45
$4500
Finn "Oneeye" Flint.


Insert:
1
$0.01
"Poor Boy" McCoy
11 22
3 8
13
45
1
$0.01
"Poor Boy" McCoy
3
$200
"Shiny Buckles" Scott
8
$450
"Toothy Grin" Skewers
11
$600
Wylie "Timbers" Scar
13
$342
Jake "Voodoo" Murphy
22
$12.99
"Pretty one" Prout
45
$4500
Finn "Oneeye" Flint.


Insert:
10
$800
“Silver Tooth” Shiver
11
8
22
3
10
13
45
1
$0.01
“Poor Boy” McCoy
3
$200
“Shiny Buckles” Scott
8
$450
“Toothy Grin” Skewers
10
$800
“Silver Tooth” Shiver
11
$600
Wylie “Timbers” Scar
13
$342
Jake “Voodoo” Murphy
22
$12.99
“Pretty one” Prout
45
$4500
Finn “Oneeye” Flint.

13 — k-D Trees

Learning Objectives

After studying this chapter, a student should be able to:

- explain what a *range search* is;
- define the properties of a k-dimensional tree (k-D tree);
- understand the relationship between 1-D trees, ordered binary trees, and AVL trees; and
- explain how to perform range searches in 1-D and 2-D trees.

13.1 Range Searching

In this chapter we will introduce k-dimensional trees (or just k-D trees). k-D trees are excellent for solving a problem known as *range search*.

A simple example of range search would be: "find all of the elements in a collection with keys between a and b". This is an example of one-dimensional range search (there is only one range of values to consider).

A more complicated example of a range search would be: "find all of the customers of a business who are between 21 to 40 years of age, who purchased products between January 1 and June 30, 2013, and who have spent between $200 and $1000 on products". This is an example of a three-dimensional range search, because each element returned from the collection has to meet three criteria — three pieces of data from the element have to be within certain ranges.

A more generic example of a range search would be: "find all of the points (a,b) such that $a_{min} \leq a \leq a_{max}$ and $b_{min} \leq b_{max} \leq b_{max}$". This is an example of a two-dimensional range search, but we can easily see how this could be generalized to three dimensional (a,b,c), 4 dimensional (a,b,c,d) or even n-dimensional points $(a_1,a_2,a_3,\ldots a_n)$ (for any $n > 0$).

k-dimensional trees are an excellent solution to range search problems when the collection of data elements in which we are searching is fixed or is only updated once in a while. This is because it is relatively easy to build a nearly perfectly balanced k-dimensional tree from a collection of data elements, but, once built, it is relatively hard to maintain the tree's balance when inserting or removing individual elements. The k-dimensional tree is not well-suited for range searching on data

that is changing in real-time. It might, however, be suitable for a customer database where we can tolerate the tree being rebuilt, say, once per day.

13.2 1-D Trees for One Dimensional Range Search

It turns out that you already know about 1-D trees. Ordered binary trees and AVL trees are 1-D trees, they just use different methods (if any) for balancing the tree. Suppose a collection of integers is stored in an ordered binary tree. Range searching in a 1-D tree is similar to searching for a single element. Note that each internal node in such a tree contains an element, and the value of that element determines which ranges of values can be found in the subtrees. Suppose we are looking for the integers between 25 and 42 (our *query range*) and the root node contains the value 15. We can immediately conclude that all of the elements that would be in the query range must be in the right subtree because everything in the query range is bigger than 15. If the root node instead contained the value 50, we could immediately conclude that any elements within the query range must be in the left subtree. If, however, the root node contained the value 37, we would have to conclude that there could be elements within the query range in both the left **and** right subtrees. This process would then continue recursively, in either one or both of the root node's subtrees.

More generally, if the query range is $[a,b]$ (all elements between a and b, inclusive), and the element in an internal node p is less than a, then any elements in the subtree rooted at p that lie within the range must be in the right subtree of p. If the element stored in p is larger than b, then any elements in the subtree rooted at p that lie within the query range must be in the left subtree of p. If the element at p is within the query range, then there could be elements in the query range in both subtrees of p. What follows is an algorithm that will find all elements in the query range $[a,b]$ in an ordered binary tree or AVL tree (which are, by definition, 1-D trees).

```
Algorithm searchRange(T, a, b):
Search for elements between a and b (inclusive) in the
ordered (1-D) tree T.  Returns the set of in-range elements in T.

if T is empty
    return the empty set
else if T.rootItem() < a
    // any in-range elements are in right subtree
    return searchRange(T.rightSubtree, a, b)
else if b < T.rootItem()
    // any in-range elements are in left subtree
    return searchRange(T.leftSubtree, a, b)
else
    // in-range elements could exist in both subtrees.
    // L and R are the set of in-range elements in the
    // left and right subtrees of T, respectively.
    L = searchRange(T.leftSubtree, a, b)
    R = searchRange(T.rightSubtree, a, b)

    // Return the set union of L, R, and rootItem()
    return SetUnion( L, R, T.rootItem() )
```

13.3 2-D Trees

A 2-D tree is a k-D tree with $k = 2$. It stores elements whose keys are two dimensional points (x, y). Note that x and y need not be integers, they could be any data items of any type that are comparable and can be placed in an order. For ease of presentation, we will show examples where the elements are points on the Cartesian plane. Just like 1-D ordered binary trees or AVL trees, each internal and leaf node of a 2-D tree store a key-element pair. In our upcoming illustrations of 2-D trees, we show only the keys for brevity.

A 2-D tree works like a combination of two 1-D ordered binary trees, one for the x dimension and one for the y dimension. At even levels of the tree (levels 0, 2, 4, etc.), the branch is based on the x-coordinate; at odd levels of the tree the branch is based on the y-coordinate. This is illustrated in Figure 13.1. The colour of each node corresponds to the line of the same colour in the Cartesian plane shown below the tree. Each tree node splits a rectangular region of the Cartesian plane into two sections based on either the x- or y-coordinate of its key, depending on its level. The root node splits the entire plane into a part where $x < 4$ and a part where $x \geq 4$. Then the green node splits the $x < 4$ region into subregions where $y < 7$ and $y \geq 7$ respectively. Similarly the red node splits the $x \geq 4$ region into a $y < 2$ subregion and a $y \geq 2$ subregion. This process continues recursively down the tree, alternating on each successive level the coordinate in which the split occurs. Thus, each node in the tree represents a region of the Cartesian plane, and the coordinate stored in the node splits its region into subregions represented by its child nodes. Notice how the constraints on the bottom level of arrows in the tree match perfectly with the rectangular regions of the Cartesian space as partitioned by the coloured lines. The algorithm for a 2D range search follows.

```
Algorithm searchRange2D(T, xmin, xmax, ymin, ymax, depth):
T - subtree in which to search for elements between a and b (inclusive).
xmin, xmax - lower and upper bounds of search range for first dimension
ymin, ymax - lower and upper bounds of search range for second dimension
depth - level of the root of subtree T in the overall tree.
Returns the set of in-range elements in T.

if T is empty
    return the empty set

if( depth mod 2 == 0)              // is depth even?
        splitValue = T.rootItem().x
        min = xmin, max = xmax
else
    splitValue = T.rootItem().y
    min = ymin, max = ymax

if splitValue < min             // in-range elements are in right subtree
    return searchRange2D(T.rightSubtree, xmin, xmax, ymin, ymax, depth + 1)
else if max < splitValue        // in-range elements are in left subtree
    return searchRange2D(T.leftSubtree, xmin, xmax, ymin, ymax, depth + 1)
else
    // in-range elements could exist in both subtrees.
    L = searchRange2D(T.leftSubtree, xmin, xmax, ymin, ymax, depth + 1)
    R = searchRange2D(T.rightSubtree, xmin, xmax, ymin, ymax, depth + 1)
    if both coordinates of T.rootItem() are in range
        return SetUnion( L, R, T.rootItem() )
    else
        return SetUnion( L, R )
```

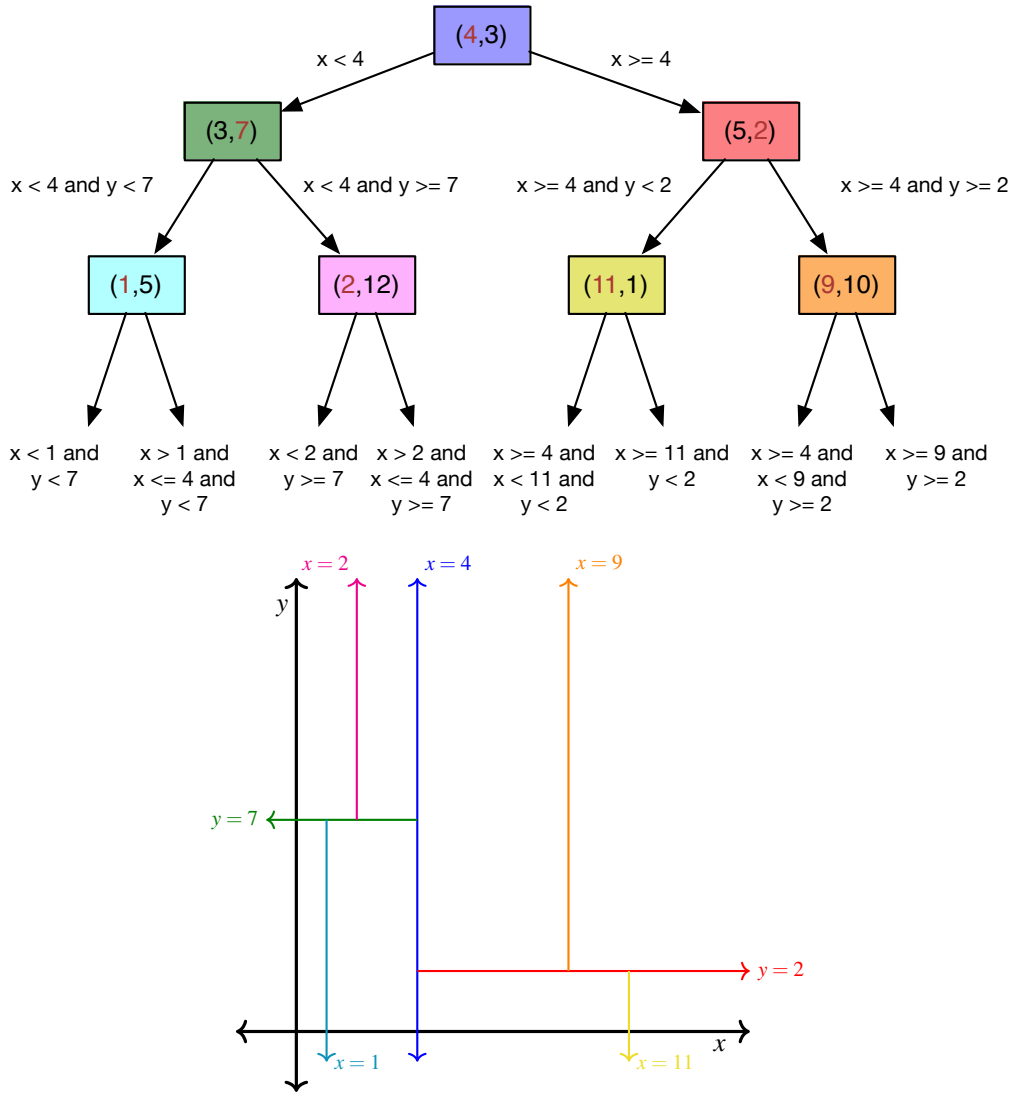


Figure 13.1: A 2-D tree showing the 7 keys in the top three levels. The red coordinate at each node is the coordinate used to determine the branching. Even levels branch on the x-coordinate, odd levels branch on the y coordinate (the root is at level 0, as per our convention). Arrows from the bottom-most nodes are labeled with the constraints on the points that would appear in each lower subtree. The colours of each node correspond to the lines in the Cartesian plane pictured below the tree. Each node splits a rectangular region of the Cartesian plane into two subregions.

13.3.1 2D Ranges

A 2-D range search for keys with x in the range $[x_{min}, x_{max}]$ and y in the range $[y_{min}, y_{max}]$ is equivalent to search for all the keys in the tree which fall into the 2D rectangle with lower left corner (x_{min}, y_{min}) and upper-right corner (x_{max}, y_{max}). The bottom of Figure 13.2 shows the query rectangle for $x_{min} = 3$, $x_{max} = 7$, $y_{min} = 9$ and $y_{max} = 11$ drawn in grey. The search for points in the grey rectangle, as usual, starts at the root of the tree. Since the blue line is within the rectangle, we could have points on either side of the blue line in the tree, so we have to search both children of the blue node (green, and red). Since the green line is completely below the query rectangle, we only need to search above it, so we only need to consider the right child of the green node (the pink node). Continuing down the tree, since the query rectangle is to the right of the pink line we only need to consider the right child of the pink node (shown by the red arrow); and the recursion would continue down from there. Upon returning from processing the green node, we would then process the red node. Since the query rectangle is completely above the red line we only need to look at the right child of the red node (the orange node). The query rectangle is completely to the left of the orange line, so we only have to look at the left child of the orange node (again, denoted by the red arrow); again the recursion continues down the tree until we find all in-range keys.

Notice how the cyan and yellow nodes are not visited because they represent subregions of Cartesian space that do not contain the rectangle. The cyan node represents everything to the left of the blue line and below the green line, while the yellow node represented everything to the right of the blue line and below the red line. The query rectangle does not intersect either of those regions, which also means that no children of the cyan or yellow nodes need to be visited, because they represent subregions of regions that we already know do not contain the query rectangle. For similar reasons only the right child of the magenta (pink) node and the left child of the orange node are visited. The left child of the magenta node represents everything above the green line and to the left of the magenta line which does not contain the query region. Finally, the right child of the orange node represents the region that is above the red line and to the right of the orange line which, again, does not contain any part of the query rectangle.

So we can see that the range search algorithm only examines parts of the tree which could possibly contain keys in the desired range. If the range is relatively small compared to the extremes of each coordinate in all keys, then only a small fraction of the tree needs to be searched; this is what makes range searching efficient. Of course, it is quite possible to specify a query range that contains most, or all of the keys in the tree. In that case most, or all of the nodes of the tree would be visited in the search. In class, we will try to convince you that the time complexity of a range search is (almost) proportional to the number of keys in the tree that fall within the query range.

13.4 Higher-dimensional k-D Trees

3D trees, and higher dimensional trees, just extend and generalize the workings of a 2D tree. For a 3D tree, level 0 splits on the x-coordinate, level 1 splits in the y-coordinate, and level 2 splits on the z coordinate. Then level 3 splits on the x-coordinate again, and so on. In a 3D tree, each node represents a rectangular cuboid volume of 3D Cartesian space, and the appropriate coordinate represents a plane that splits the rectangular cuboid into two rectangular cuboidal sub-volumes.

In general, for a k-dimensional tree, the coordinate used to split at a given depth is $(depth \bmod k) + 1$ where depth is the level of the current node. Thus, on level 12 of a 5-D tree, the split would be based on the coordinate of the $12 \bmod 5 + 1 = 2 + 1 = 3 =$ 3rd dimension. Note that if we were storing the dimensions of a point in an array, we might want to drop the "+1", because this would

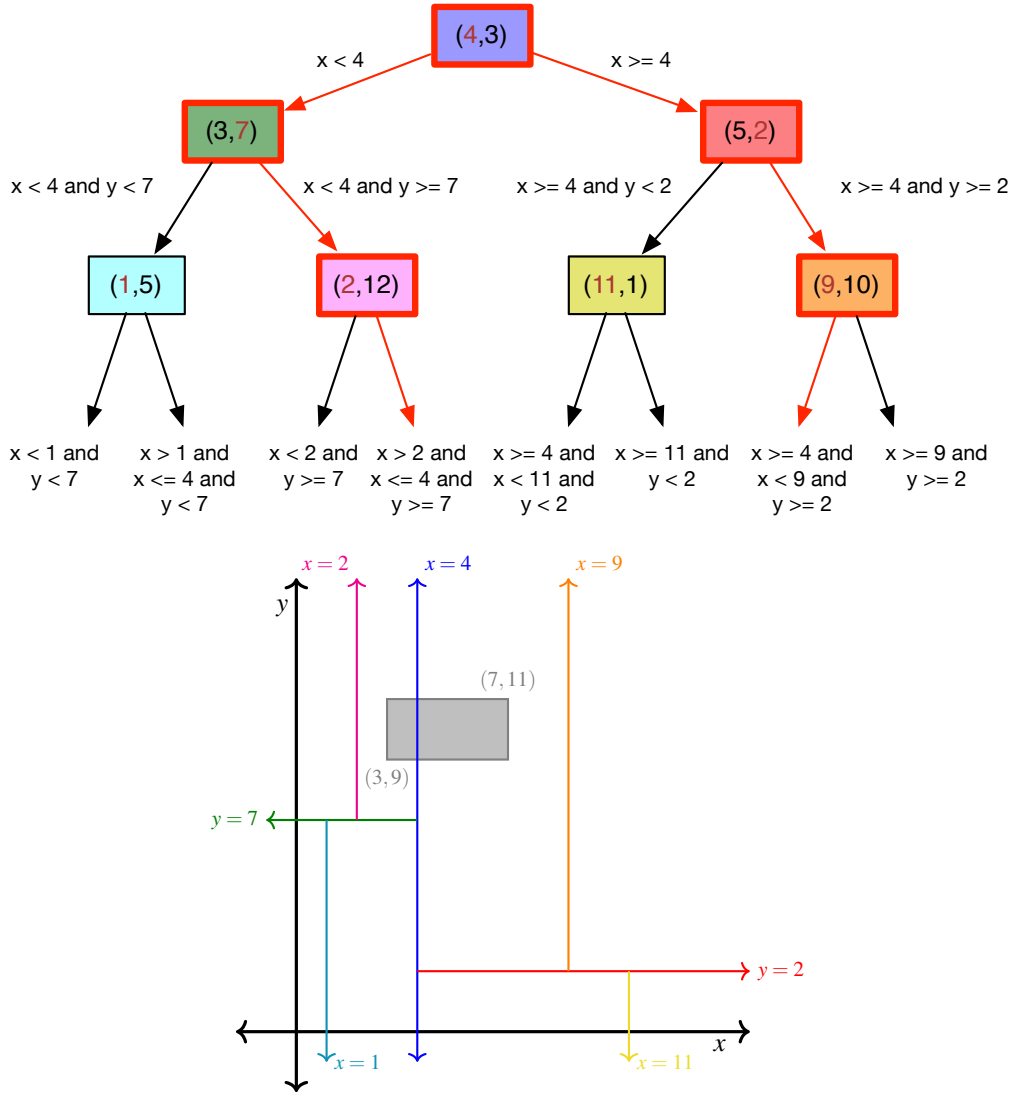


Figure 13.2: The same 2-D tree shown in Figure 13.1. Below the tree, the query range defined by $x_{min} = 3$, $x_{max} = 7$, $y_{min} = 9$ and $y_{max} = 11$ is shown shaded in gray. The nodes of the tree bordered in red are the nodes visited by the recursive range search algorithm `searchRange2D`, above.

give us the **offset** of the dimensions within the array. The offset of the dimension at level 12 of the tree would be offset $12 \bmod 5 = 2$ (the offset of the array in which the third dimension of the point is stored).

13.5 Building k-D Trees

k-D trees are built from a set of k-dimensional keys (and their elements) in such a way that the tree is as close to being perfectly balanced as possible. We will leave the algorithm and implementation of this as a subject for a programming assignment.

13.6 Inserting and Deleting Elements from an Existing k-D Tree

In this course we will not consider the algorithms for inserting and removing individual elements from an existing k-D tree. They are quite non-trivial and are beyond the scope of this course. Because of the complexity of individual insertions and deletions, k-D trees are more suited to the workflow of: build once, use it for a while, then after a period of time completely re-build the tree to reflect addition or removal of elements.

14 — Other Trees

Learning Objectives

After studying this chapter, a student should be able to:

- recognize and distinguish between the types of trees described in this chapter and in previous chapters;
- have a basic understanding of the properties of the trees described in this chapter;
- be able to (conceptually) build a trie tree for a set of keys;
- be able to identify if a trie tree contains a given key; and
- recall particular applications that are suited to the specific types of trees.

In this chapter, we will give some descriptions of some other kinds of trees, without going into too much detail about the algorithms for insertion and deletion. Instead we will focus more on their proprieties and the applications to which they are suited.

14.1 B+ Trees

B+ Trees are a generalization of the 2-3 trees from Chapter 12. In a B+ tree of order b, each node has between $\lceil b/2 \rceil$ and b children (except the root which may have between 2 and b children). Like 2-3 trees, leaves in a B+ tree store key-element pairs (or links from keys to elements) and internal nodes contain up to $b-1$ keys that partition the ranges of keys that can be found in the subtrees rooted at each child. B+ trees also typically add additional links between siblings at the last level of internal nodes to facilitate accessing the elements in the tree sequentially in order of their keys; the leaf nodes are linked together like a linked list from left to right. Aside from this addition, the 2-3 trees that we previously studied are nearly identical to B+ trees of order 3.

Insertion and deletion from B+ trees are just a generalization of the operations for 2-3 trees. Upon insertion, if a node needs to have more than b children, then it is split, and the new node must be accommodated by the split node's parent, which may, in turn, cause other splits. If deletion results in a key having fewer than $\lceil b/2 \rceil$ children, then stealing or giving will occur similar to the way it occurs when deleting from a 2-3 tree.

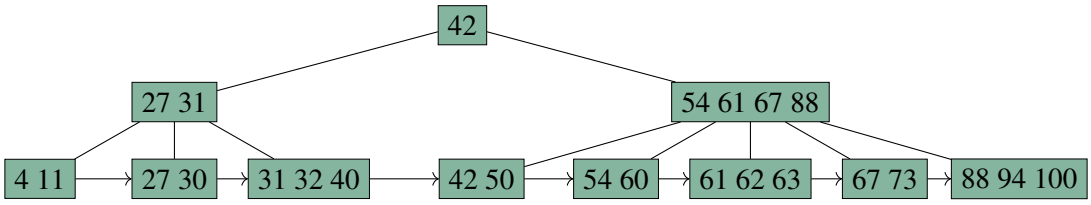


Figure 14.1: A B+ Tree of order 5.

B+ trees are used a lot in the implementation of filesystems and databases. This is because in these applications the data elements (and the tree itself) may be stored in data blocks on a disk rather than in RAM. Because high-order B+ trees have a very high fanout, fewer I/O operations (which are relatively very expensive) are needed versus an implementation that uses a binary tree structure.

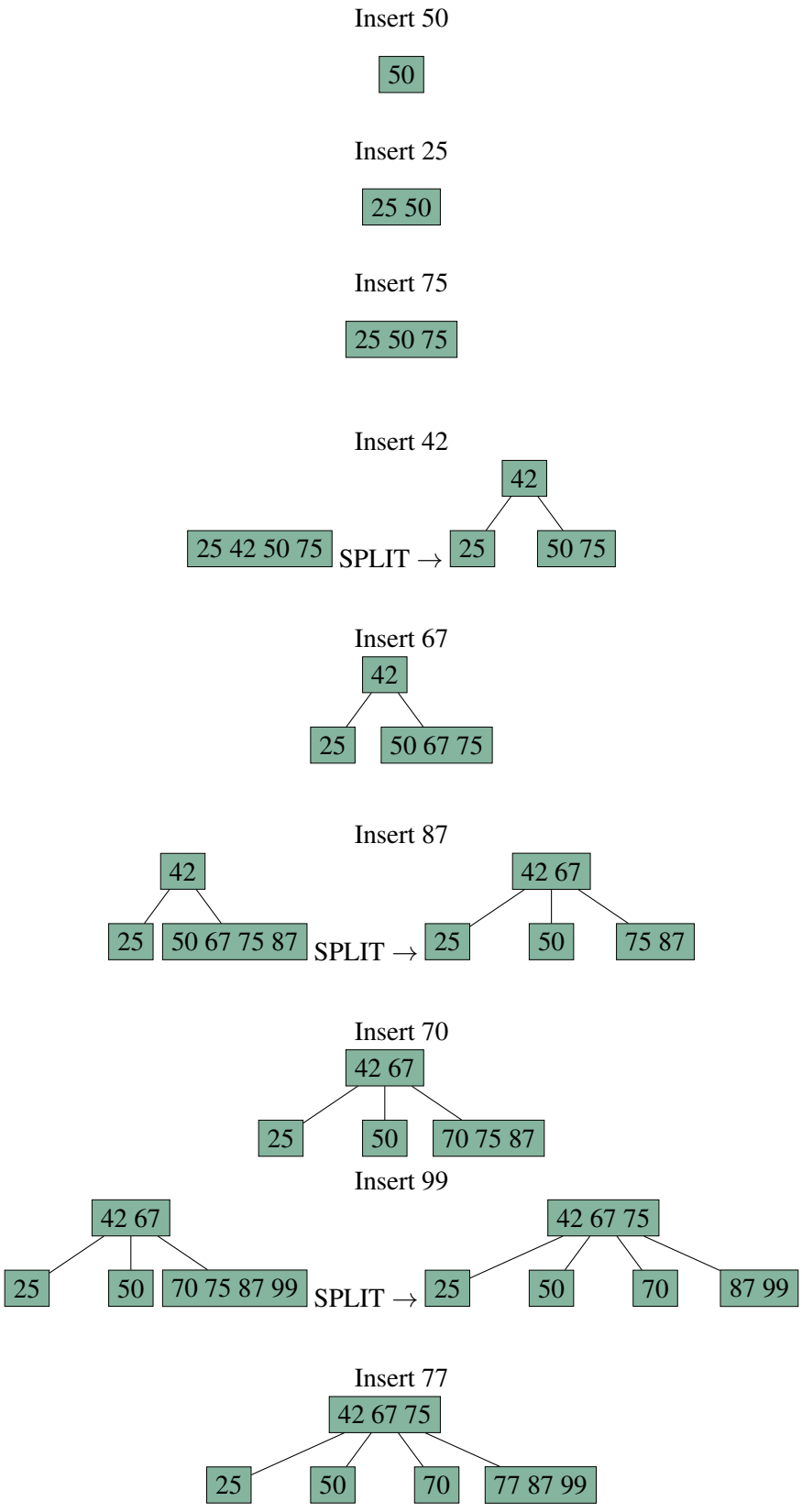
Several filesystems use B+ trees for metadata indexing including NTFS, ReiserFS, NSS, XFS, JFS, and ReFS. B+ trees are also used in many relational databases (where, again, data is stored on disk) such as IBM DB2, Informix, Microsoft SQL Server, Oracle 8, Sybase ASE, and SQLite to record table indices. Source: Wikipedia.

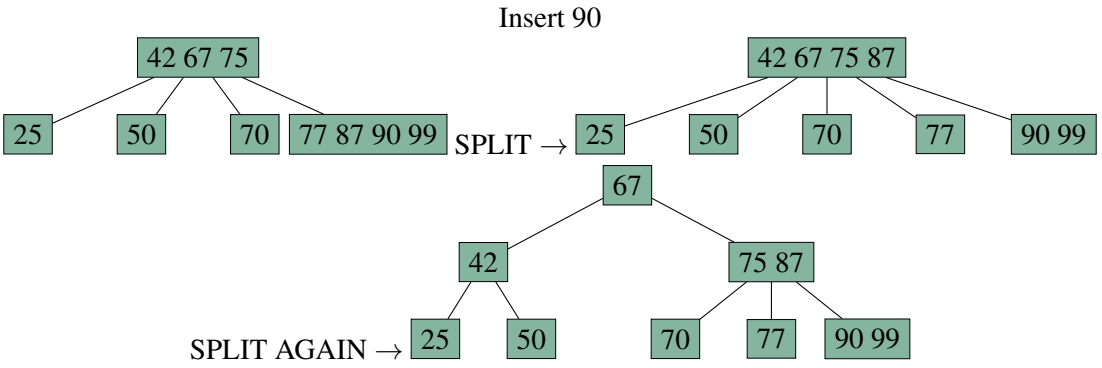
14.2 B Trees

B-trees are generalizations of ordered binary trees. They are very similar to B+ trees, except that in B-trees, not all key-element pairs are at the leaves (just like in an ordered binary tree!). Instead of storing just keys, internal nodes store key-element pairs, but the keys of those elements **also** serve as the partitioning keys for the ranges of keys in the subtrees of the node's children, just as the keys in the internal nodes of a 2-3 tree do. Nodes in a B-tree of order b have between b and $2b$ children. Insertions use a form of splitting similar to B+ trees, deletions use stealing and giving similar to B+ trees. This ensures that the tree is balanced; all leaf nodes in a B-tree are at the same level (even though not all elements are stored in leaves).

The main advantage of the B+ tree over a B-tree is that, since all of the elements are at the leaves in a B+ tree, it facilitates much better sequential access to the elements (because they can just be read from left-to-right along the leaf level).

■ **Example 14.1** Here we show a sequence of insertions into a B-tree of order 2 where every internal node has between 2 and 4 children. For brevity, we only show keys, but **every** key shown, even the ones in internal nodes, is associated with an element as well. Insertion works just like in an ordered binary tree (recurse to leaf level, insert at appropriate spot) except that we can store more than one key-element pair in a node. When necessary, we split nodes, much like for 2-3 trees, except that one key-element pair gets promoted to the parent of the split node to act as a new partitioning key.


Insert 50
50
Insert 25
25 50
Insert 75
25 50 75
Insert 42
25 42 50 75
SPLIT →
42
25
50 75
Insert 67
42
25
50 67 75
Insert 87
42
25
50 67 75 87
SPLIT →
42 67
25
50
75 87
Insert 70
42 67
25
50
70 75 87
Insert 99
42 67
25
50
70 75 87 99
SPLIT →
42 67 75
25
50
70
87 99
Insert 77
42 67 75
25
50
70
77 87 99



■

Like B+ trees, B-trees are useful for disk-based data structures because the high branching factor minimizes disk accesses. The maximum number of children is usually governed by the disk block size, which, in turn, governs how many key-element pairs can fit in a disk block. Thus, each node is associated with a disk block, and each node visited in a search operation causes one disk access. The high branching factor reduces the height of the tree, which keeps the number of disk accesses needed to search for an element very small.

As observed before, a B+ tree is more advantageous to use when sequential access to elements in key-order is preferred because only leaf nodes need to be accessed for such an operation on a B+ tree. On the other hand, the B-tree would be preferred for random accesses since, on average, random searches would be slightly faster (by a constant factor) in a B-tree because of the possibility of finding a desired element before reaching the bottom level of the tree.

Both B-trees and B+ trees have an $O(\log n)$ worst-case time complexity for searching, insertion, deletion where n is the number of key-element pairs in the tree.

14.3 Red-Black Trees

Red-black trees are yet another flavour of self-balancing trees. Every red-black tree is an ordered binary tree and must have the usual properties of ordered binary trees. But red-black trees also have the following additional properties:

1. Each node is assigned a colour, either "red" or "black".
2. The root node is black.
3. All leaf nodes have no contents and are black.
4. Every red node must have two black child nodes.
5. Every path from a starting node p to any of its descendants must have the same number of black nodes.

Searching red-black trees is identical to searching an ordered binary tree because red-black trees are just a special case of ordered binary trees. The insertion and deletion algorithms, however, are quite complicated. The insertion algorithm, for example, follows the normal insertion algorithm for ordered binary trees to find the location to insert the new element. Then the new element is added in that position in a red node (with two empty black nodes as children). What happens next depends on the colour of the newly inserted node's parent, the inserted node's uncle (sibling of parent) and the inserted node's grandparent. There are 5 different possibilities (which we will not go into here).

In a red-black tree, the length of the longest path from root to leaf is guaranteed to be no more than twice the length of the shortest path from root to leaf (but the overall height is still $O(\log n)$.

Since red-black trees are less rigidly balanced than AVL trees, red-black trees require re-balancing less often than AVL trees. AVL trees, being more rigidly balanced, will generally result in faster search times because the overall height of AVL trees will be smaller on average. AVL trees, therefore, are well-suited for data structures that may be built once, or only once in a while. Red-black trees are more well-suited to data structures that will change very often because less time will be spent on rebalancing compared to AVL trees. Red-black trees are used in the Linux kernel's Completely Fair Scheduler where it stores data on processes (running programs) keyed by the amount of time they have spent executing on the CPU. The process in the left-most node (which has had the least amount of execution time) is removed from the tree, allowed to run on the CPU for a while (on the order of milliseconds), then returned to the tree, usually in a different position because it just got more execution time. Such a red-black tree gets updated very very frequently, so insertions and deletions need to be as efficient as possible, while still maintaining reasonable search time. The red-black tree fulfills these requirements well.

14.4 Trie Trees

A trie tree is an m-ary tree, usually with m fairly large. The term "trie" comes from the word *retrieval* and is pronounced "try." Trie trees work under the assumption that key values can be subdivided into "characters" from some alphabet, for example, dividing a string into characters, or a number into digits. Each internal node can have up to $m = size(alphabet) + 1$ children. An array is usually used to store these children so that the i-th child can be accessed quickly.

In a trie tree, each node at level k of the trie corresponds to a prefix of length k of a key. The presence or absence of a child indicates the presence or absence of a key in the trie with that prefix. Figure 14.2 shows a trie containing the integer keys 412, 294, 417, 842, and 4129. Since the keys are integers, the alphabet consists of the digits 0 through 9, plus a special character '$'. We can tell that the trie in Figure 14.2 contains the key 4129 because the root node's '4' child is non-null. This child represents the key prefix 4. This node has a '1' child, which represents the key prefix 41, and that node has a '2' child, which represents the key prefix 412. The '412' node has a '$' reference that is non-null, which tells us that the key 412 is not only just a prefix of a key in the tree, but actually **is a key** in the tree. Note that the '4' child of the root has a null '$' reference; this tells us that while 4 is the prefix of one or more keys, it is not, itself, a key in the trie. Now the '412' node also has a '9' child, which represents the key prefix 4129. The '4129' node has a non-null '$' reference, which tells us that 4129 is an actual key in the trie. Observe that the trie could also store key-element pairs if we change the '$' references so that they refer to the elements belonging to the key that corresponds to the node.

We can reduce the number of nodes needed in a trie. For links to subtrees that contain only one actual key, we can alter the link to point directly to that key. This is called a *compact trie*. Figure 14.3 shows a compact trie that is equivalent to the trie pictured in Figure 14.2.

Trie trees are useful as general dictionaries, that is, containers in which items can be inserted, deleted, and looked up. Tries offer outstandingly fast implementations of these operations with the tradeoff being that they consume absurdly large amounts of memory space. The time complexity for the search, insertion, and deletion operations on tries are $O(length(key) + 1)$. That is, the time required to find, insert, or remove a key depends **only** on the length of the key! It doesn't matter whether there are ten keys or a million keys in the trie, a search for a key of length 5 still only requires that we examine at most 6 nodes. We'll answer the question of space complexity of a trie tree in lectures. For now, think about how much space it would take to store all alphanumeric strings

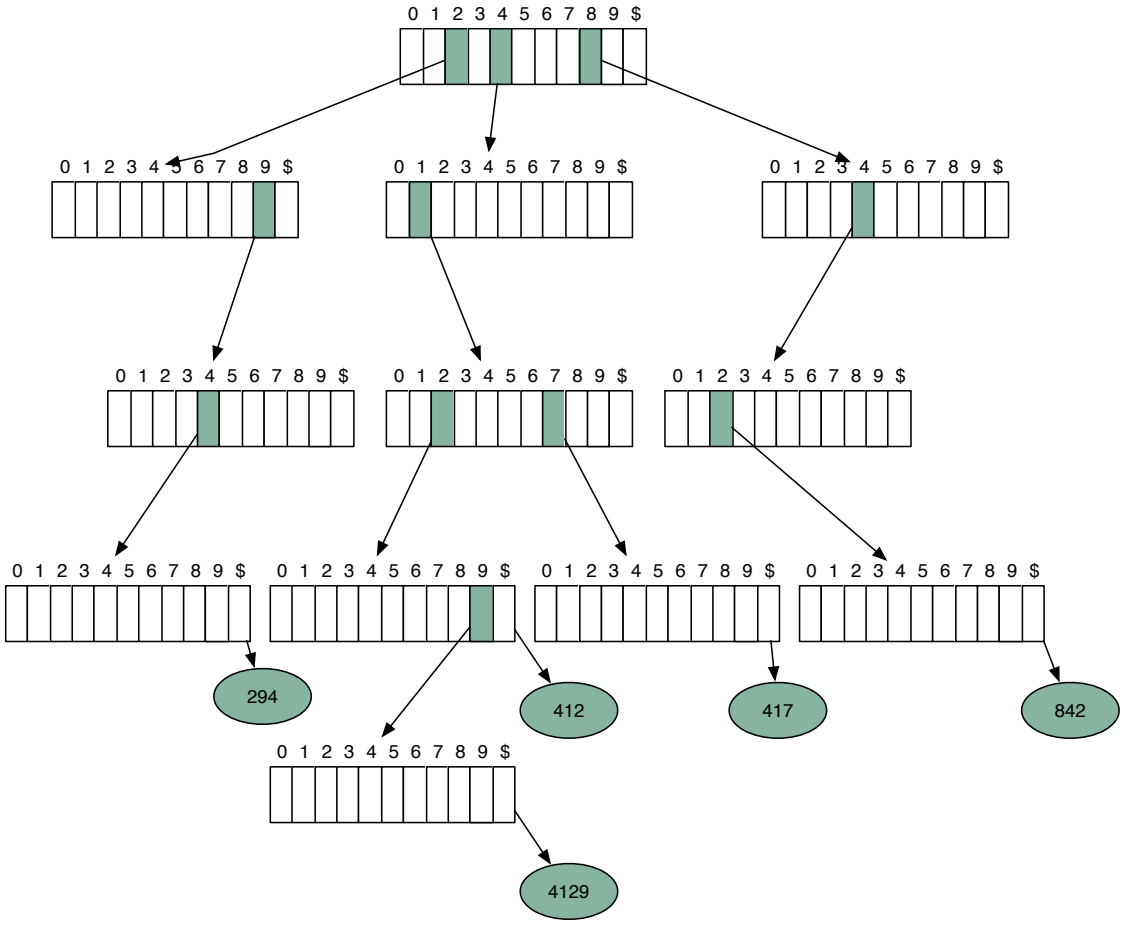


Figure 14.2: A trie for integer keys containing the keys 412, 294, 417, 842 and 4129.

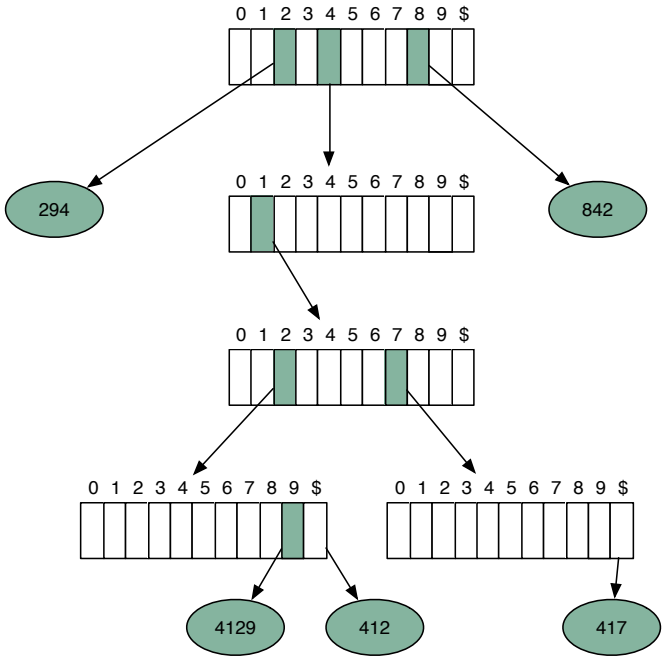
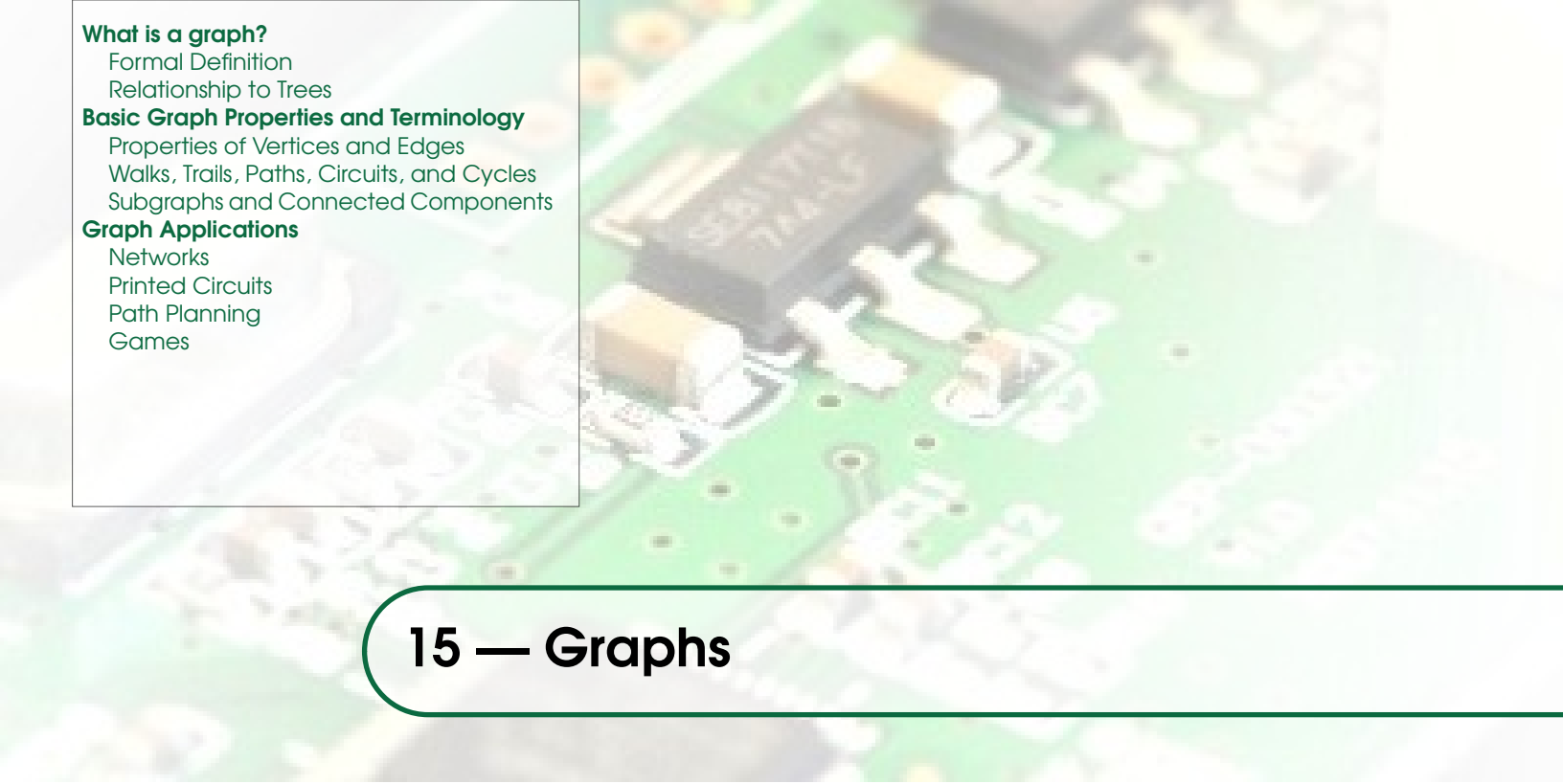


Figure 14.3: A compact trie for integer keys containing the keys 412, 294, 417, 842 and 4129.

of length 5.



15 — Graphs

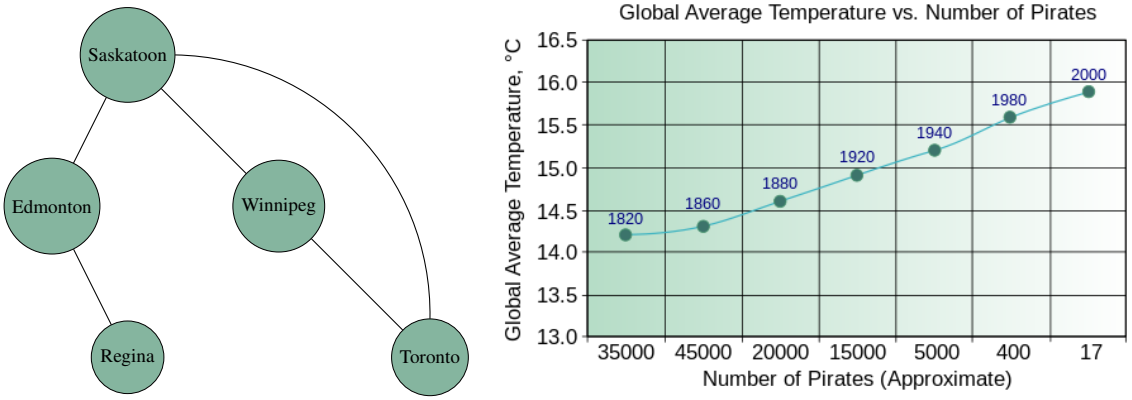
Learning Objectives

After studying this chapter, a student should be able to:

- explain what a graph is, both intuitively and mathematically;
- distinguish between *directed* and *undirected* graphs;
- determine the *degree* of vertices in an undirected graph, and the *indegree* and *outdegree* of vertices in a directed graph;
- define and distinguish between *walks*, *trails*, *paths*, *circuits*, and *cycles* of a graph;
- explain what it means for vertices in a graph to be *connected* or *strongly connected*;
- define and explain what a *subgraph* of a graph is;
- identify the *connected components* of an undirected graph and the *strongly connected components* of a directed graph; and
- give some examples of applications of graphs.

15.1 What is a graph?

In the context of data structures, a graph is a representation of a mathematical relation. It records which elements of a set S are related to each other in some fashion. For example, a graph could be formed from a set of cities and knowledge of direct airplane flights between those cities. The graph would record which cities have direct flights between them. Visually, a graph is represented by drawing a labeled circle for each element of the set, and drawing lines between the circles representing set elements that are related, for example, Figure 15.1(a). Such a graph might represent that there are direct flights between, for example, Saskatoon and Winnipeg, and Saskatoon and Toronto, but not between Sasktoon and Regina, or Regina and Winnipeg. When we talk about graphs, we do **not** mean line graphs (like the one pictured in 15.1(b), or bar graphs, or pie charts.



(a) (b)

Figure 15.1: Graphs: (a) a graph data structure (the good kind of graph); (b) the other kind of graph (Image by Mikhail Ryazanov from Wikimedia Commons; reproduced under Creative Commons Attribution-Share Alike 3.0 license). When we talk about graphs in the context of data structures, we mean the kind of graph in (a).

15.1.1 Formal Definition

We now proceed to a more formal definition of graphs.

Definition 15.1 A *graph* $G = (V, E)$ consists of a set V of *vertices* (or *nodes*) and a set $E \subseteq V \times V$ of *edges* (or *arcs*).

For example, the visual depiction of the graph in Figure 15.1 is mathematically represented, according to Definition 15.1, as: $G = (V, E)$ where

$$\begin{aligned} V &= \{Saskatoon, Edmonton, Winnipeg, Regina, Toronto\} \\ E &= \{(Saskatoon, Toronto), (Toronto, Saskatoon), (Saskatoon, Winnipeg), (Winnipeg, Saskatoon), \\ &\quad (Winnipeg, Toronto), (Toronto, Winnipeg), (Saskatoon, Edmonton), (Edmonton, Saskatoon), \\ &\quad (Edmonton, Regina), (Regina, Edmonton)\} \end{aligned}$$

V is the set of vertices (cities in this case), and E is a set of ordered pairs of related vertices (in this case representing direct flights between cities). The reason we need symmetric pairs, e.g. both $(Saskatoon, Toronto)$ and $(Toronto, Saskatoon)$ is because the graph's edges are non-directional. Saskatoon is related to Toronto, and Toronto is related to Saskatoon in this particular relation. Such a graph is called an *undirected graph*. For brevity we may sometimes use *unordered pairs* to denote undirected edges, which are denoted with square brackets, for example, $[Saskatoon, Toronto]$. The ordering doesn't matter; this notation denotes both that Saskatoon is related to Toronto, and that Toronto is related to Saskatoon. The above set of edges could then be rewritten more briefly as:

$$\begin{aligned} E &= \{[Saskatoon, Toronto], [Saskatoon, Winnipeg], [Winnipeg, Toronto], \\ &\quad [Saskatoon, Edmonton], [Edmonton, Regina]\}. \end{aligned}$$

More formally, if for all $(x, y) \in E$, there is (y, x) in E, for all vertices $x, y \in V$, then the graph $G = (V, E)$ is an *undirected graph*. If a graph does not have this property it is called a *directed graph*

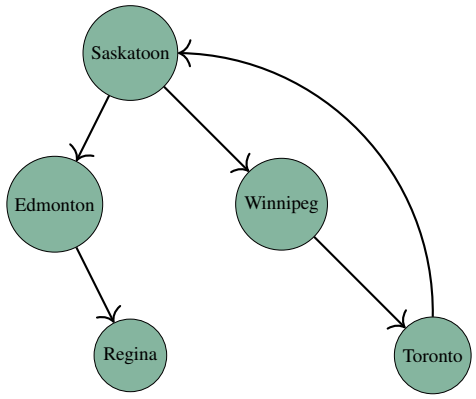


Figure 15.2: A directed graph. The edges end with arrows to denote direction.

(sometimes written *digraph*) and its visual representation is drawn such that the edges have arrows to indicate the direction of the connection, like in Figure 15.2. We would interpret this graph to mean that there are direct flights from Toronto to Saskatoon, and from Saskatoon to Winnipeg, but there are *not* direct flights from Saskatoon to Toronto or from Winnipeg to Saskatoon (the arrows only go in one direction!). The mathematical representation of this graph would be:

$$\begin{aligned} V &= \{Saskatoon, Edmonton, Winnipeg, Regina, Toronto\} \\ E &= \{(Toronto, Saskatoon), (Saskatoon, Winnipeg), (Winnipeg, Toronto), \\ &\quad (Saskatoon, Edmonton), (Edmonton, Regina)\}. \end{aligned}$$

Note the use of ordered (not unordered) pairs. Because the set of edges is not symmetric (we have $(Toronto, Saskatoon)$, but not $(Saskatoon, Toronto)$) the graph is directed rather than undirected. Undirected graphs are a special case of directed graphs. We generally assume that a graph is directed unless it is explicitly stated that a graph is undirected.

15.1.2 Relationship to Trees

One may consider that graphs are a generalization of trees, or, conversely, that trees are special cases of graphs. A graph may have cycles (a set of edges that form a continuous loop) while a tree may not. Every tree is also a graph, but not every graph is a tree.

15.2 Basic Graph Properties and Terminology

We begin with some properties of vertices and edges.

15.2.1 Properties of Vertices and Edges

For a directed edge (a, b), a is called the *initial* or *source* vertex, b is the *terminal* or *destination* vertex.

The *indegree* of a vertex v in a graph, written $indegree(v)$, is the number of directed edges for which v is the destination vertex. The *outdegree* of a vertex v in a graph, written $outdegree(v)$ is the number of directed edges for which v is the source vertex. In an undirected graph, the edge $[a, b]$ is

said to be *incident* on both a and b. The *degree* of a vertex v in an undirected graph is the number of incident edges on v.[1] Two vertices are said to be *adjacent* if there is an edge between them.

A vertex v may be related to itself, as is allowed in reflexive relations. In such a case, the edge (v,v) appears in the set of edges E and is sometimes called a *self-loop*. A self-loop might appear in a graph where the vertices represent Java classes, and edges represent the "calls" relation, that is, the edge (a,b) is in E if class a calls a method in class b. If a method in class x calls another method in class x (as Java classes often do) then the edge (x,x) would appear in such a graph. Note that a self-loop adds 1 to both the indegree and the outdegree of a node in a directed graph, and adds 2 to the degree of a vertex in an undirected graph (both ends of the edge are incident on the vertex).

15.2.2 Walks, Trails, Paths, Circuits, and Cycles

Frequently we are interested in more than the direct relationships between vertices. For example, consider again the example of a directed graph where vertices represent cities and edges represent direct flights. Suppose we want to answer the question of whether it is possible to get from city a to city b by taking more than one flight. To answer this type of question, we need some additional terminology.

A *walk* is an alternating sequence of vertices and connecting edges of a graph, that is, a route from vertex to vertex along connecting edges. A walk can begin and end on the same vertex and can visit any vertex, or travel any edge any number of times. In a directed graph, an edge can only be crossed in one direction, from the source vertex to the destination vertex.

A *trail* is a walk that does not travel across the same edge more than once. A trail may visit the same vertex more than once, but this must occur via different incoming and outgoing edges each time. A trail that begins and ends on the same vertex is called a *circuit*.

A *path* is a walk that does not visit any vertex more than once except that the first and last vertex on the path may be the same. A path that starts and ends on the same vertex is called a *cycle*.

The *length* of a walk/trail/path/circuit/cycle is the number of edges in the walk/trail/path/circuit/-cycle.

A vertex a is *connected* to another vertex b if there is a path from a to b. Note that in a directed graph with vertices a and b, it is possible that there is a path from a to b, but not from vertex b to vertex a.

Having defined these concepts, it should be clear that the answer to the question posed at the beginning of this section (is there a series of direct flights that can be taken to get from city a to city b?) can be solved by answering the question "is vertex a connected to vertex b?".

15.2.3 Subgraphs and Connected Components

A *subgraph* of a graph $G = (V,E)$ is a graph $G' = (V',E')$ such that:

- $V' \subseteq V$; and
- if $e = (u,v) \in E$, and $u,v \in V'$, then $e \in E'$.

More intuitively, a subgraph of a graph G is any graph formed by removing zero or more vertices from G as well as all edges that had a removed vertex as an endpoint.

A subgraph $G' = (V',E')$ of an **undirected** graph G is called a *connected component* of G if every pair of vertices in V' are connected and there is no larger subgraph of G that contains all of the vertices in V' that also has this property. A graph G may have more than one connected component,

[1] Sometimes it is said that a directed edge is incident on its destination vertex, in which case the indegree of a vertex v is equal to the number of directed edges incident on v.

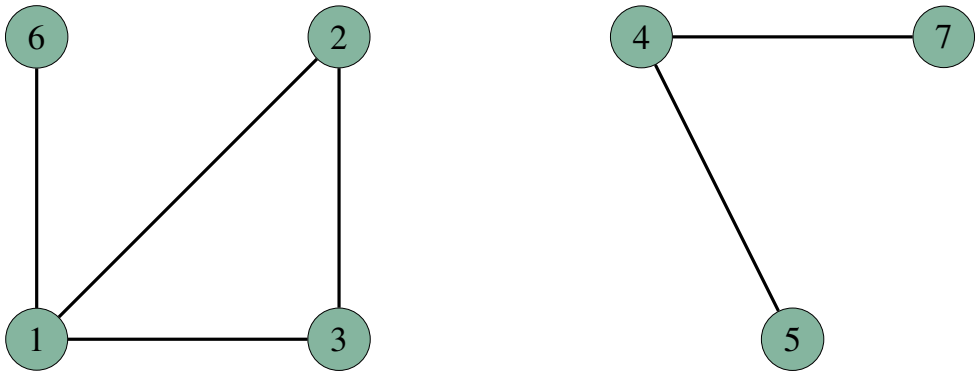


Figure 15.3: An undirected graph with two connected components.

as illustrated in Figure 15.3. Note that this figure does not depict two graphs; it is a single graph with vertex set $V = \{1,2,3,4,5,6,7\}$. The subset of V, $V_1 = \{1,2,3,6\}$ is a connected component of the graph because every pair of vertices in V_1 is connected, and no superset of V_1 has this property. Likewise for the subset of V, $V_2 = \{4,5,7\}$. However, the vertex set $V_3 = \{1,6,2\}$ is not a connected component. While it is true that each pair of vertices in V_3 is connected, V_1 is a superset of V_3 that also has this property, so V_3 is not, by definition, a connected component in this graph.

The notion of a connected component as defined above doesn't really work for directed graphs. In a directed graph, v_1 may be connected to v_2, but that doesn't guarantee that v_2 is connected to v_1. Therefore, we need a stronger condition to define something analogous to a connected component for a directed graph. Recall from Section 15.2.2 that vertices v_1 and v_2 are *connected* if there is a path from v_1 to v_2. Two vertices v_1 and v_2 in a **directed** graph are said to be *strongly connected* if v_1 is connected to v_2 **and** v_2 is connected to v_1.

A subgraph $G' = (V', E')$ of a **directed** graph $G = (V, E)$ is a *strongly connected component* of G if every pair of vertices in V' is strongly connected, and no other subset of V containing V' also has this property. The directed graph shown in Figure 15.4 has three strongly connected components. The first is the set of vertices $\{1,2,3,6\}$. Every pair of vertices in this set is strongly connected, and there is no superset of vertices in V that have this property. The second strongly connected component is the set $\{4,7\}$. The third is the set $\{5\}$. Although every pair of vertices in the set $\{1,2,3\}$ are strongly connected, it is not a strongly connected component because the set $\{1,2,3,6\}$ is a superset of $\{1,2,3\}$ in which every pair of vertices is strongly connected. Note that vertex 8 is not part of **any** strongly connected component because there is no vertex that is strongly connected to 8, not even itself!

15.3 Graph Applications

Graphs can be used to represent and model a wide range of important abstract and concrete entities. In this section we present just a few examples.

15.3.1 Networks

Graphs are especially good at representing networks of relationships such as links between web pages on the internet, friendships on Facebook, followers on Twitter, connectivity of the internet

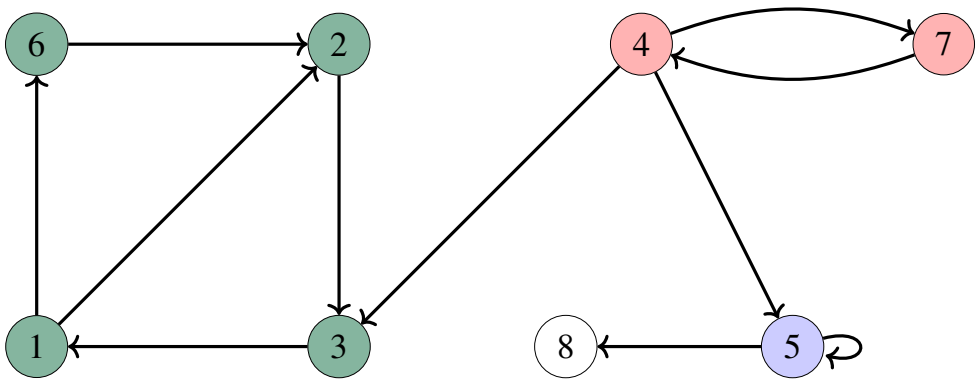


Figure 15.4: A directed graph with three strongly connected components. Each component is shaded in a different colour. Node 8 is unshaded because it does not belong to any strongly connected component.

(routers and links between them), and, as we saw before, travel links between cities. Entities that can be related are vertices in the graph (e.g. web pages, Facebook users, Twitter users, internet routers, cites), and edges indicate that a relationship exists (e.g. one web page links to another, a Facebook user is friends with another, a Twitter user follows another, two internet routers are connected, there is a direct flight from one city to another).

Once we have a graph representation of a network, we can ask many interesting questions; here are a few examples:

- How many friends does a Facebook user have on average?
  (What is the average outdegree of a vertex?)
- What is the average number of clicks needed to get from one web page to another?
  (What is the average length of the shortest path between a pair of vertices? You might be surprised how small this number is for the graph of web page connectivity!)
- What is the minimum number of internet routers that have to fail before some part of the internet gets cut off from the rest of it?
  (What is the minimum degree of a vertex?)
- Given a set of internet routers, and an integer k, what is the smallest number of connections between routers needed to ensure that the minimum degree of a vertex is k? (There's a linear time algorithm for this!)

15.3.2 Printed Circuits

In this application, junctions on an electronic circuit board design are vertices, and there is an edge between vertices if there is to be an electrical link between them. Since these links are actually going to be manufactured as physical links, a circuit board must be designed so that no links between electrical junctions cross. So given a graph of vertices representing electrical junctions and connections between them, we can ask the question: can the graph be drawn so that no edges cross? A graph which can be drawn such that no edges cross is called a *planar graph*. So the question of whether connections can be printed on a circuit board so they do not cross is equivalent to "is the electrical connectivity graph planar?"

Figure 15.5 shows a graph on the left; it is not clear from the way it is drawn whether it is planar.

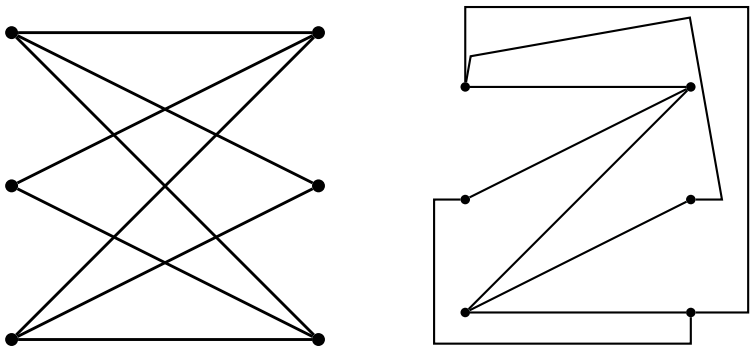


Figure 15.5: Left: planar graph. Right: the same graph, redrawn to prove that it is planar. If the middle pair of vertices is connected, then the graph is no longer planar — it becomes impossible to draw it with no crossing edges. This would represent an impossible-to-manufacture circuit board.

On the right, we see the same graph drawn differently, proving that it is planar. If you look closely, you'll see that the set of edges is identical, just drawn differently. However, if we add just one additional edge connecting the middle pair of vertices, it becomes impossible to draw without any crossing edges. Such a graph would represent a circuit that cannot be manufactured. There is a linear-time algorithm for determining whether a graph is planar.

15.3.3 Path Planning

Video games often use graphs to model how characters in the game can move through a three-dimensional virtual world. In this section we explain a simplified version of how this might work.

Imagine a graph where the vertices represent coordinates in the 3D game world that characters and/or monsters are allowed to occupy, and we connect vertices with an edge only if the 3D geometry of the game world allows travel in a straight line between the locations represented by those two vertices. Some vertices might not be connected because of an intervening wall or other obstacle.

Now, as the game plays in real-time, the game AI might decide that a shambling zombie has to move from a position represented by graph vertex v_1 to another location represented by graph vertex v_2 in order to try to eat the hero's brains. At that point in time, the system can use an algorithm (maybe a shortest-paths algorithm, we'll cover such algorithms later in some detail) to find the best way for the zombie to travel through the graph to get to the player. But suppose that part-way through moving towards v_2 the conditions in the game world change — maybe the hero has seen the zombie coming and moved to run away, or to close in for a killing shotgun blast. At this point, the game AI might change the zombie's destination and a new path must be planned out. This could happen many times per second.

15.3.4 Games

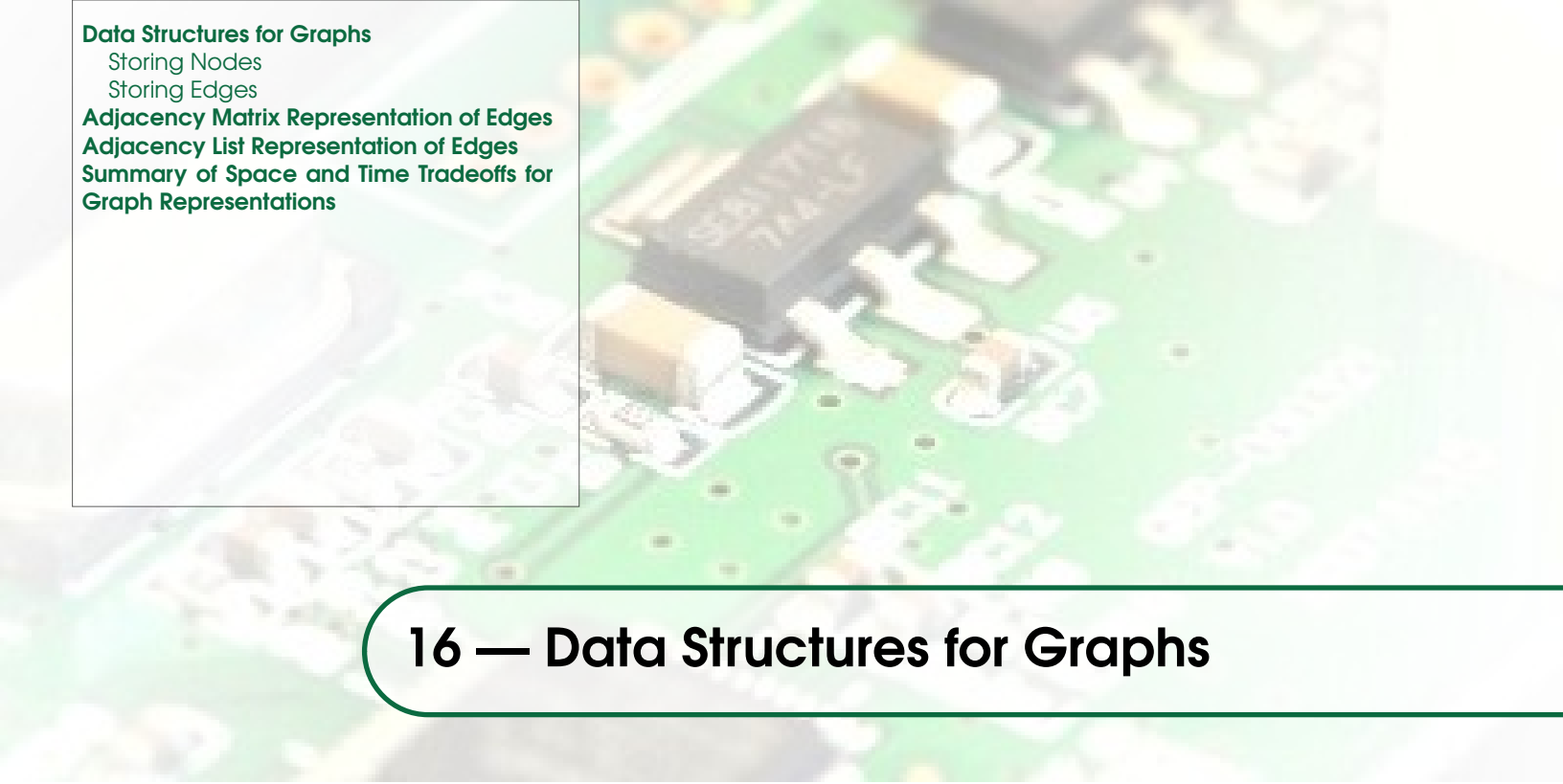
Have you ever played the game "Six degrees of Kevin Bacon"?[2] In this game, one player names a Hollywood movie actor. The other player then has to connect that actor to the actor Kevin Bacon in as few links as possible. Connections are made between actors using the "appeared in a movie

[2]This game is a play on "Six Degrees of Separation" which is a theory that everyone in the world is connected to everyone else by no more than six steps of "introduction" (i.e. the "friendship" relation).

with" relationship. For example, given the actor Sean Connery, a correct response would be: "Sean Connery was in *The Rock* with David Bowie who was in *A Few Good Men* with Kevin Bacon." An actor's Bacon number is the smallest number of connections which relates them to Kevin Bacon. Therefore, Sean Connery's Bacon Number is at most 2, and David Bowie's Bacon number is at most 1.

This game is played with Kevin Bacon as the target, because he has appeared in so many films with so many actors, that it is believed that the average Bacon number over all actors is quite low; indeed it has been estimated to be about 3. However, according to a list at the Oracle of Bacon website, there are many actors that are better connected. The only reason this list can exist is because of graphs. A graph is built with a vertex for all actors, and undirected edges link vertices representing actors who have been in a movie together. The connectivity of an actor is found by solving the so-called "single-source shortest paths" problem on the graph. This is the problem of choosing a vertex in the graph, and finding the shortest paths from that vertex to every other vertex. Then we compute the average length of those paths. If we choose Kevin Bacon as the starting vertex, then we get the average Bacon number. If we chose Christopher Lee (Count Dooku, in Star Wars, Saruman in Lord of the Rings), we would get the average Lee number (average number of links from an actor to Christopher Lee) which we would find is actually less than the average Bacon number (it's about 2.89). The Oracle of Bacon's list could be generated by solving the "single-source shortest paths" problem for each actor's starting vertex. Interestingly, at the time of writing, there are 395 actors and actresses who are better connected than Kevin Bacon.

For interested parties, the main Oracle of Bacon web page will generate solutions to the game when given a starting actor. You may also be interested in the page about how the Oracle of Bacon works which explains a little about the graph theory and algorithms behind the website.



16 — Data Structures for Graphs

Learning Objectives

After studying this chapter, a student should be able to:

- draw a visualization (nodes and edges) of a graph from an *adjacency matrix* representation;
- determine the *adjacency matrix* representation of a graph from its visualization;
- draw a visualization of a graph from an *adjacency list* representation;
- determine the *adjacency list* representation of a graph from its visualization; and
- explain the advantages and disadvantages of the *adjacency list* and *adjacency matrix* representations in terms of space requirements, and time requirements for the *random access* and *sequential access* patterns.

16.1 Data Structures for Graphs

Now that we have reviewed what a graph is, and some of its basic properties, we are ready to talk about how to build a data structure to represent a graph. To make our discussions about graphs easier, we will now make a simplifying assumption about the names of nodes in a graph.

Suppose we have a graph $G = (V, E)$ with $|V| = n$ (i.e. there are n nodes in the graph). Further suppose that we assign to each node in V a unique integer index between 0 and $n-1$. In essence, we are just renaming all of the nodes in V to unique numbers between 0 and $n-1$. We can do this renaming without loss of generality as long as the new node names are unique. Thus, in the rest of this chapter we will assume that $V = \{0, 1, 2, \ldots, n-1\}$ for all graphs.

16.1.1 Storing Nodes

Most graph algorithms tend to require access to nodes in a graph in a more or less random order rather than sequentially. Therefore, we must have the ability to access a specific node quickly. Usually the number of nodes in a graph either doesn't change, or an upper bound is available. Therefore, it is not unreasonable to store the nodes of a graph in an array. In Java, we might create a node object, and then store the nodes in the graph in an array of that object's type.

Storing graph nodes in an array allows us to access a specific node i in the graph in constant time, which is perfect for the random access patterns to nodes that we anticipate. We can find a desired node by indexing into the array that stores the nodes using the index of the desired node.

16.1.2 Storing Edges

In a graph, edges don't have names. A desired edge is usually accessed by specifying its source and destination nodes. There are two very common types of access patterns that one finds in algorithms that operate on graphs.

The first typical edge access pattern is: *given two vertices, access the edge between them (if it exists)*, that is, a request to access a specific edge. This amounts to a *random access pattern* similar to the access pattern we anticipate for nodes. Thus, it would be wise to adopt a mechanism for storing edges in which a specific edge can be accessed quickly.

The second typical edge access pattern is: *given a vertex, access its outgoing edges one-by-one*. This is a *sequential access pattern*. Such a pattern might occur in a traversal of the graph, in which having visited a vertex, the algorithm must then visit all of that vertex's immediate neighbours. Thus, it would be wise to adopt a mechanism for storing edges that facilitates efficient sequential access to all outgoing edges from a given vertex.

Alas, as frequently occurs in computer science, we are now faced with a tradeoff. Fast random access and fast sequential access to outgoing edges of a given node turn out to be competing goals. There are two common ways to represent the edges of a graph. These are the *adjacency matrix* and *adjacency list* representations. We shall see that the adjacency matrix favours the random access pattern, while the adjacency list favours sequential access pattern. We will also see that the adjacency list representation requires less space most of the time. We devote the next two sections of this chapter to these two representations, respectively.

16.2 Adjacency Matrix Representation of Edges

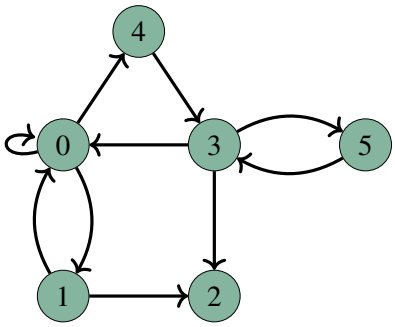
Under the assumption that $V = \{0, 1, 2, \ldots, n-1\}$, the *adjacency matrix* A for a directed graph $G = (V, E)$ is defined to be:

$$A(i,j) = \begin{cases} 1 & \text{if } (i,j) \in E; \\ 0 & \text{otherwise,} \end{cases}$$

where $A(i,j)$ is the entry of the matrix A at row i, column j. In other words, adjacency matrix entry $A(i,j)$ is a 1 if and only if there is an edge from node i to node j. Figure 16.1 shows an example of a directed graph and its adjacency matrix. Entry $(3,2)$ (row 3, column 2) has a 1 in it because there is an edge from node 3 to node 2. On the other hand, entry $(2,3)$ (row 2, column 3) has a 0 because there isn't an edge from node 2 to node 3.

Undirected graphs are represented by requiring that the adjacency matrix be symmetric about the diagonal. If this matrix is symmetric, then it means that if entry (i,j) is 1, then entry (j,i) is also 1. This means that the graph always has both (or neither) edges (i,j) and (j,i) for any i and j, which is the definition of an undirected graph.

It is easy to see how the matrix representation can be implemented — we can use a two dimensional array. In Java, we might have a 2D array of boolean to represent the connections between nodes. This is fine if we don't have any information to record about edges besides the edge's endpoints. But sometimes we might want to associate additional data with an edge. So another approach to implementing adjacency matrices in Java is to create an edge object, and then have a 2D



	0	1	2	3	4	5
0	1	1	0	0	1	0
1	1	0	1	0	0	0
2	0	0	0	0	0	0
3	1	0	1	0	0	1
4	0	0	0	1	0	0
5	0	0	0	1	0	0

Figure 16.1: A graph and its adjacency matrix.

array of those objects such that entry (i, j) in the array is non-null, and references an edge object only if there is an edge from node i to node j.

If we use a 2D array to implement an adjacency matrix, then we can see that random access to edges is very fast — we can index the array using the indices of the nodes of the edge's endpoints and gain access to a specific edge in constant time. However, consider sequential access. Suppose we want to access all of the outgoing edges from node i. The only way to make sure we find all such edges is to examine **every** entry in the i-th row of the adjacency matrix. This will require $O(n)$ time, where n is the number of nodes in the graph, **regardless** of how many actual outgoing edges there are from node i. We can do better than this with adjacency lists, but we have to sacrifice the constant random-access time.

Another factor to consider is the amount of storage required for edges. For the adjacency matrix we have to store a value for every entry (i, j) in the matrix, whether there is actually an edge between (i, j) or not. There are n^2 such entries. Thus, the storage requirements for an adjacency matrix is $O(n^2)$ no matter how many edges there are. Therefore we can make the stronger statement that the space requirements for an adjacency matrix is $\Theta(n^2)$.

16.3 Adjacency List Representation of Edges

In the adjacency list representation of edges, we construct an array of linked lists, one for each node. The i-th linked list is a list of all of the destination vertices for edges for which i is the source vertex. In other words, if there is an edge (i, j), then node j appears in the i-th linked list. Figure 16.2 shows the same graph as in Figure 16.1 along with its adjacency list representation. Since the graph contains the edges $(1,2)$ and $(1,0)$, the nodes 2 and 0 appear on the adjacency list for node 1.

Undirected graphs are realized by adjusting the implementation so that it requires that whenever node j appears in the adjacency list for node i, node i appears on the adjacency list for node j.

In a Java implementation, the list for vertex i might contain references to the node objects that are the destination nodes of edges outgoing from vertex i.

Let's consider now the time and space tradeoffs of the adjacency list representation compared to those of the adjacency matrix representation. Access to a specific edge (the random-access pattern) is not constant time. In order to find a specific edge (i, j) we must, in the worst case, search every node on the adjacency list for node i. This takes $O(outdegree(i))$ time in the worst case (which is when the edge we are looking for does not exist). Clearly the adjacency matrix wins if we anticipate having more random accesses to edges than sequential accesses. On the other hand, to find all of the

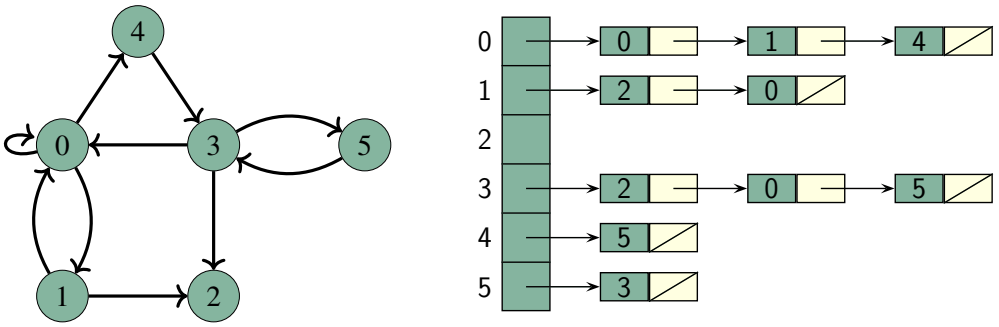


Figure 16.2: A graph and its adjacency lists. There is an array of lists represented here by the green vertically-oriented array. The list at index i of the array of lists contains the destination vertices of all edges that are outgoing from node i.

edges outgoing from a specific node i, we simply traverse the adjacency list for node i. This always requires $O(outdegree(i))$ time, so we can say that sequential access to all outgoing edges of a node i is $\Theta(outdegree(i))$. This is an improvement over sequential access for the adjacency matrix (which requires $\Theta(n)$ time) when $outdegree(i)$ is less than n. For most graphs, the outdegree of any given node is very often much smaller than n, so the adjacency list representation is a clear winner if we anticipate having more sequential accesses to edges than random accesses.

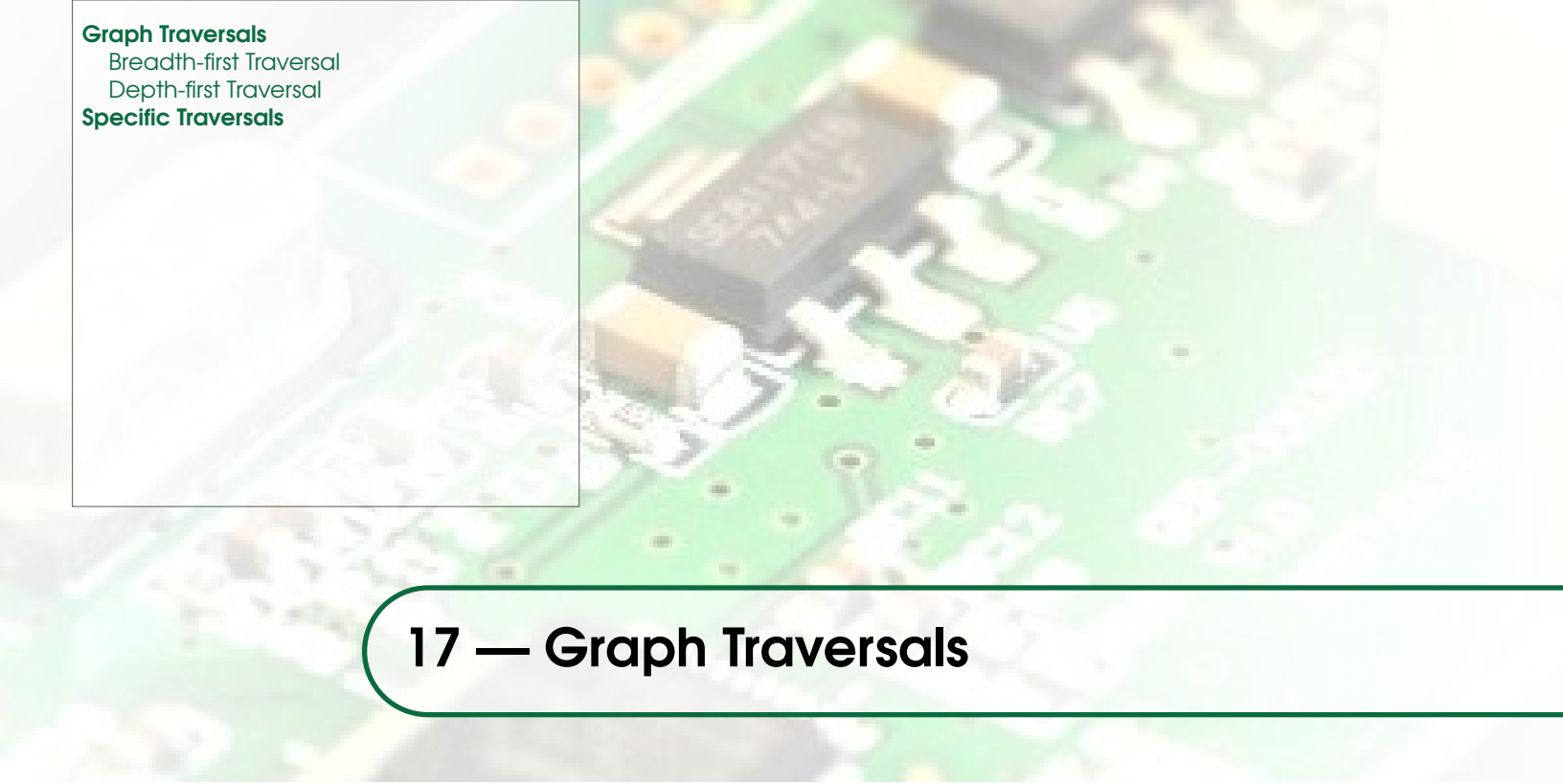
What about the space requirements? Let $m = |E|$ be the number of edges in the graph. Unless every possible edge in the graph exists, then $m < n^2$. In the adjacency list representation, there is $O(n)$ space required by the array of lists. Then there is one node in a list for each directed edge in the graph. Thus, the total number of nodes in **all** of the adjacency lists is exactly m. Therefore, the space required by the adjacency list implementation is $\Theta(n+m)$. If every possible edge exists, then $m = n^2$ and the space requirements are about the same as the adjacency list. More typically, m will be much smaller than n^2, resulting in a significant savings over the $\Theta(n^2)$ space required by the adjacency matrix. This savings can be very significant for large n.

16.4 Summary of Space and Time Tradeoffs for Graph Representations

To close this chapter, we present a summary of the time and space tradeoffs between the two graph representations.

Representation	Sequential Edge Access	Random Edge Access	Space
Adjacency Matrix	$\Theta(n)$	$\Theta(1)$	$\Theta(n^2)$
Adjacency List	$\Theta(outdegree(i))$	$O(outdegree(i))$	$\Theta(n+m)$

In the above table, $n = |V|$ is the number of nodes, $m = |E|$ is the number of edges, and i is the source vertex of the edge(s) being accessed.



17 — Graph Traversals

Learning Objectives

After studying this chapter, a student should be able to:

- explain and distinguish between the strategies of breadth-first vs depth-first traversal;
- find a valid *breadth-first traversal* of a given graph; and
- find a valid *depth-first traversal* of a given graph.

17.1 Graph Traversals

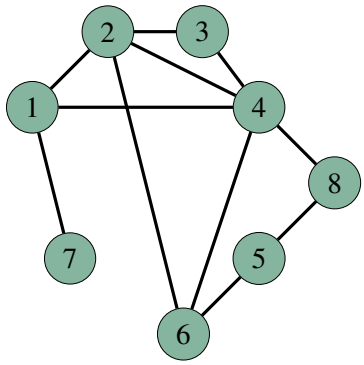
Graph traversals are algorithms for visiting the nodes of a graph in a systematic order. In this respect they are similar to list or tree traversal algorithms which also visit every node or element. For graphs, there are two basic strategies for making traversals:

- if the current node is v, visit each node directly adjacent to v before considering nodes that are further away; and
- if the current node is v, recursively visit an adjacent node and all nodes further away before considering the next node adjacent to v.

These two strategies correspond to breadth-first and depth-first graph traversals, respectively. The algorithms will somewhat resemble the breadth- and depth-first traversal algorithms for trees because trees are special cases of graphs. The main differences you will find in the graph traversal algorithms are caused by the need to deal with the fact that, unlike trees, graphs may contain cycles which could cause a traversal to arrive back at an already-visited node.

17.1.1 Breadth-first Traversal

To begin a breadth-first traversal, a starting node is selected, and is visited. Recall that "visiting" a node means to perform some relevant operation with or on any data associated with the node. The breadth-first traversal then visits all nodes that are reachable by a path of length 1 from the start node. Once these are exhausted, nodes that are reachable via a path of length 2 from the start node are visited, and so on, until all nodes have been visited. Figure 17.1 provides an example. In the



Shortest Path Length from Node 1	Nodes
0	1
1	2, 4, 7
2	3, 6, 8
3	5

Figure 17.1: Breadth-first traversal of a graph. Node 1 is the start node. It is visited first. Then all nodes reachable by a path of length 1 from the start node are visited. Then all nodes reachable by a path of length 2 from the start node are visited, and so on.

pictured graph, we are using node 1 as the start node for the traversal. The start node is visited first. Then all nodes reachable by a path of length 1 from the start node must be visited. If there is more than one such node, they may be visited in any order. After all such nodes have been visited, nodes not already visited that are reachable by a path of length 2 from the start node are visited. Again, these nodes may be visited in any order. Once these nodes are exhausted, we visit all unvisited nodes that are reachable by a path of length 3 from the start node, and so on.

It is important to note that for most graphs, the order in which nodes are visited in a breadth-first traversal is not unique. Nodes at the same "distance" from the start node can be visited in any order, as long as they are all visited before nodes that are more distant. In the case of the graph in Figure 17.1, there are 36 different possible valid breadth-first traversals,[1] a few of which are given in the following list:

- 1, 7, 4, 2, 3, 6, 8, 5
- 1, 4, 2, 7, 6, 3, 8, 5
- 1, 2, 4, 7, 8, 6, 3, 5

We have coloured the nodes according to the length of their shortest path back to the start node. We can rearrange the order of the nodes of one colour and still have a valid breadth-first traversal, but we cannot change the order of the colours.

Another way to look at the breadth-first traversal is to find the shortest paths from the start node to each other node, and then remove any edges that do not appear on any such shortest path. The result will be a subgraph which is a tree! The tree is called the *shortest path tree*. A breadth-first traversal traverses only the subgraph corresponding to the shortest path tree, which is always traversed in level-order. The shortest path tree of the graph in Figure 17.1 is shown in Figure 17.2.

[1] 1 possible order for visiting nodes at distance 0, $3! = 6$ possible orders for visiting the nodes at distance 1, $3! = 6$ possible orders for nodes at distance 2, and 1 order for nodes at distance 3; giving $1 \times 6 \times 6 \times 1 = 36$ possible breadth-first traversal orders.

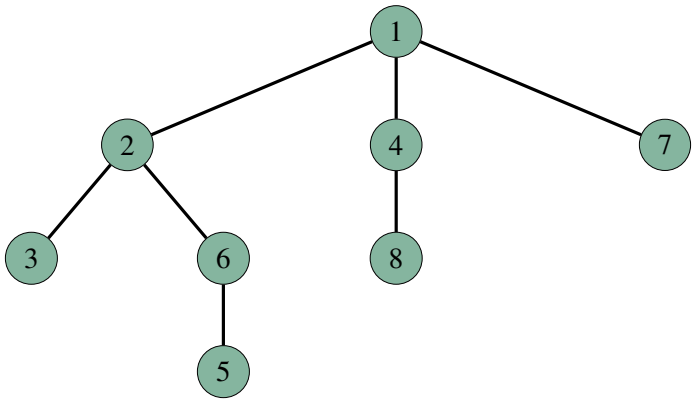


Figure 17.2: The shortest path tree for the graph in Figure 17.1. Note how the nodes on each level of the tree match the sets of nodes in the table in Figure 17.1 and the coloured groups in the examples of breadth-first traversal sequences in the text.

The algorithm for breadth-first traversal is given below. Like the breadth-first traversal of trees, it requires a queue. We will talk more about this pseudocode in class, and follow along with it as we do some examples.

```
// An algorithm for breadth-first traversal of a graph
Algorithm bft(s)
s is the start node in the graph

q = new Queue()

For each vertex v of V
    reached(v) = false

reached(s) = true
q.insert(s)

while not q.isEmpty() do
    w = q.item()    // get top node on stack
    q.deleteItem()  // pop the stack

    // At this point, we would perform the "visit" operation on w

    // Add vertices adjacent to w to queue if not already visited
    For each v adjacent to w do
        if not reached(v)
            reached(v) = true
            q.insert(v)
```

17.1.2 Depth-first Traversal

The depth-first approach is similar to finding one's way out of a maze. Once the traversal starts down a path, it continues along the path visiting new nodes as long as possible. When a previous vertex ("dead end") is reached, we back up and take the first available alternate path. In this way, the depth-first traversal continues to try different paths until all possible paths that don't lead to already

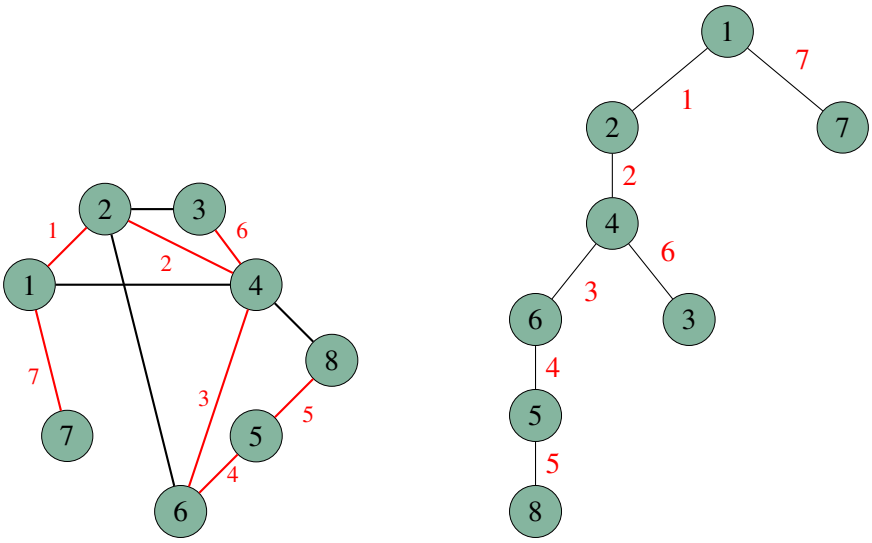


Figure 17.3: Depth-first tree of a graph. Left: a possible order in which edges are traversed in a depth-first traversal of a graph; red numbers show the order in which edges are traversed, thus, nodes are visited in the order: 1, 2, 4, 6, 5, 8, 3, 7. Right: The subgraph consisting of the red edges in the graph on the left drawn as a depth-first tree; red edge labels correspond to the same labels in the graph on the left.

visited vertices have been tried.

The left side of Figure 17.3 shows a graph where the edges followed in a possible depth-first traversal are coloured red and numbered according to the order in which they are traversed; node 1 is the start node. Notice how the path from the start node gets longer until edge number 5, at which time, there are no nodes adjacent to node 8 that haven't already been visited. The algorithm then backtracks along the path it followed until a node is found which has an unvisited adjacent vertex. In this example, it turns out to be the node 4, which is adjacent to the unvisited vertex 3. The edge to node 3 is traversed, node 3 is visited, and once again, there are no vertices adjacent to 3 that are unvisited. The algorithm backtracks again all the way back to node 1, which is the only vertex remaining that has an adjacent unvisited vertex. The edge to node 7 is traversed, and node 7 is visited. Again there are no unvisited nodes adjacent to 7 so we backtrack to node 1. Node 1 now has no unvisited adjacent nodes, so the algorithm is finished.

The right side of Figure 17.3 shows the subgraph of the left side of said figure that consists of only the red edges and drawn as a *depth-first tree*. So again, we see that, like breadth-first traversal, depth-first traversal follows the edges of a subgraph of the graph that is a tree. But note that the tree followed by a depth-first traversal it isn't a shortest-path tree as in the case of breadth-first traversal! The path in the depth-first tree from node 1 to node 4 is of length 2, but the shortest path from node 1 to node 4 is of length 1.

The algorithm for depth-first traversal of graphs, given below, is nearly identical to the algorithm for depth-first traversal of trees. We needed to add a test for whether the node we are about to visit has already been visited via another path. We also needed an initial method to set each vertex's "reached" property to false, then the familiar depth-first algorithm appears in a recursive helper method.

```
// An algorithm for depth-first traversal of a graph.
Algorithm dft(s)
s is the start node.

For each vertex v in V   // V is the set of nodes in the graph
    reached(v) = false

dftHelper(s)
```

```
// Recursive helper method for algorithm dft()
Algorithm dftHelper(v)
v is a graph node

reached(v) = true

// At this point, we would perform the visit operation for v

For each node u adjacent to v
    if not reached(u)
        dftHelper(u)
```

We will trace this algorithm in class alongside a concrete example.

Like breadth-first traversals of graphs, depth-first traversals of graphs are not unique. We make no promises about the order in which the loop `for each node u adjacent to v` processes adjacent nodes. As long as the current path from the start node is always made longer, if possible, the next node to be visited can be chosen from among the non-visited adjacent nodes in any order, and the result is still a depth-first traversal. In practice, the data structure used to store edges (adjacency list or adjacent matrix) usually imposes a convenient order.

17.2 Specific Traversals

In class we will investigate variants and specialization of the general breadth-first and depth-first traversal algorithms which will allow us to use traversals to *search* for a specific node and to use traversals to obtain the actual paths (sequence of nodes) that were travelled from the start node to each node.

18 — Graph Algorithms - Paths

Learning Objectives

After studying this chapter, a student should be able to:

- explain *Warhsall's algorithm*;
- define what a *weighted graph* is;
- explain *Floyd's algorithm*, and its similarities to and differences from Warshall's algorithm; and
- trace through *Dijkstra's algorithm* by hand showing how the *tentative distances* and *predecessor nodes* change as each graph vertex is processed.

Note: *The time complexities, and the types of graphs to which these algorithms can or should be applied to is very important. However, discussions of these things are not presented here in the pre-class readings. Instead, we discuss them in the course of doing the in-class exercises related to this topic. Do not forget to review the solutions to the in-class exercises and discussions when they are posted! Therein you will find a full discussion of the important points pertaining to time complexities and appropriate uses of the algorithms. You could also come to class to listen to and/or participate in the discussions. If you haven't been coming to class, I suggest you give it a try!*

18.1 Path Algorithms for Graphs

In the previous chapter, we saw that the breadth-first search algorithm can be modified to find the shortest path from a given start node s to a given target node t. Paths between nodes in a graph, and particularly shortest paths, are frequently of great importance in graph-based algorithms. In this chapter, you will learn about a number of classic graph algorithms which can answer questions like:

- Is there a path between two nodes s and t?
- Is there a path of a specific length between s and t?
- What is the shortest path between s and t?
- What is the shortest path between a node s and every other node in the graph?
- What is the shortest path between all pairs of nodes in the graph?

18.1.1 Number of Walks of a Specific Length Between Two Nodes

The first algorithm we will examine determines how many walks of a specific length r are between two nodes i and j in a graph. Refer to Section 15.2.2 if you need a reminder of what a walk is. Suppose the graph has n vertices.

The solution for $r = 1$ is easy. Let A be the $n \times n$ adjacency matrix of the graph in which we know that $a(i, j)$ is 1 if nodes i and j are connected by an edge. Thus A is a matrix in which each entry $a(i, j)$ is the number of walks of length 1 from i to j.

In general, we want to be able to compute a matrix A^r where each entry $a^r(i, j)$ is the number of walks of length r from nodes i to node j. A^0 is just the identity matrix (a matrix where each diagonal entry is 1 and all other entries are 0) which represents that there is a walk of length 0 from every node to itself, and no other walks of length 0. A^1 is just A, the adjacency matrix for the graph. But we want to find $A^2, A^3, \ldots, A^r$ for any r we want.

In order to see how to do this, let's consider how to construct A^2. Let's consider the possible walks of length 2 from i to j; this is the number we would need to store in entry $a^2(i, j)$ of A^2. If there is a walk of length 2 from i to j then it must pass through an intermediate node k. This walk exists only if the edges (i,k) and (k, j) are in the graph. The adjacency matrix entries $a(i,k)$ and $a(k, j)$ are equal to 1 if these edges are in the graph. This means that there is a walk from i to j **that passes through** k only if $a(i,k) \cdot a(k, j) = 1$. To say it another way, the number of walks from i to j of length 2 that pass through k is equal to $a(i,k) \cdot a(k, j)$. Now, the **total** number of walks of length 2 from i to j is equal to the sum of these products for each possible k. That looks like this:

$$\sum_{k=1}^{n} a(i,k) \cdot a(k, j).$$

Look closely at the above sum... it is exactly the product of the i-th row and the j-th column of standard matrix multiply!!! From this we can immediately conclude that the matrix multiplication of A with itself is A^2:

$$A^2 = A \cdot A$$

In general:

$$A^r = A^{r-1} \cdot A.$$

To see that this is true, we can make a similar argument as above, but this time for computing A^r from A^{r-1}. Suppose we have already computed A^{r-1}. The number of walks of length r from i to j through the intermediate vertex k is equal to the product of the number of length $r-1$ walks from i to k and the number of walks of length 1 from k to j. This is equivalent to

$$a^{r-1}(i,k) \cdot a(k, j).$$

To get $a^r(i, j)$ we need the sum of the above over all k:

$$\sum_{k=1}^{n} a^{r-1}(i,k) \cdot a(k, j).$$

and we again see that this is just the product of the i-th row of A^{r-1} and the j-th column of A in standard matrix multiplication.

In a nutshell, this section shows that the r-th power of the adjacency matrix A gives us A^r whose entries $a^r(i, j)$ tell us how many walks of length r there are from i to j. The algorithm follows.

```
// Find the number of walks in the graph of length r between
// nodes i and j
Algorithm numWalksij(i, j, r, A)
i, j - pair of nodes
r - desired walk length
A - adjacency matrix for the graph

Ar = the r-th power of A
return Ar(i,j)
```

18.1.2 Path Existence

The algorithm for *path existence* computes the answer to the question: "Is there a path from node i to node j (of any length)?" for two given nodes i and j. We can use the powers of the adjacency matrix from the previous section to answer this question. To see how, we must first establish the following claim:

> *If there is a walk from i to j of length r, then there must be a* ***path*** *from i to j of length $n-1$ or less.* (n = number of nodes in the graph)

Why is this true? Well, if $r < n-1$ the statement is trivially true. But what if $r >= n$? If this is true, then the walk contains at least n edges which means the walk contains at least $n+1$ nodes. If the walk contains at least $n+1$ nodes, then there must be some nodes that are repeated on the walk. If a node on the walk is repeated, then the walk must contain a repeating cycle, which means there must be a path from i to j from among the first $n-1$ nodes in the walk. Therefore there is a path from i to j of length $n-1$.

This observation tells us that if there are no paths from i to j of length $n-1$ or less, then there is no path from i to j of any length. We can use powers of the adjacency matrix to determine if there is a path from i to j with length $n-1$ or less. Let $B = A^1 + A^2 + \cdots + A^n$. If there is a path from i to j with length $n-1$ or less, then entry $b(i,j)$ of B will be non-zero. If there is not a path from i to j with length $n-1$ or less, then entry $b(i,j)$ will be zero. The algorithm follows.

```
// Is there a path from node i to node j?
Algorithm isPath(i, j, A)
i, j - pair of nodes
A - adjacency matrix for the graph

B = A + A^2 + A^3 + A^4 + ... + A^(n-1)
return B(i,j) > 0
```

It turns out that this algorithm is very inefficient. We will see just how inefficient in the lectures. The next section discusses Warshall's algorithm, which improves upon the algorithm from this section.

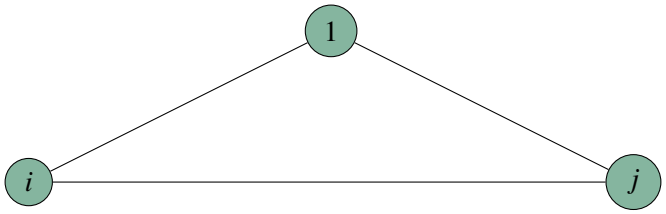
18.1.3 Path Existence — Warshall's Algorithm

Warshall's algorithm (1962) computes path existence in a faster, but less obvious fashion. Again, this algorithm operates on a graph with n nodes. To see how this algorithm works, we need to define a new kind of matrix. Let $P^{(k)}$ be a binary matrix (entries are 0 or 1 only) where $P^{(k)}(i,j) = 1$ if and only if there is a path from i to j that passes through intermediate vertices $\{1,2,\ldots,k\}$ only. That is, $P^{(k)}(i,j) = true$ only if there is a path that starts with i, ends with j, and contains no other vertex

numbered higher than k. Note that if we can compute $P^{(n)}$ then we have solved the problem because there are no nodes in the graph numbered higher than n; $P^{(n)}(i,j)$ would be true only if there was a path in the graph from i to j.

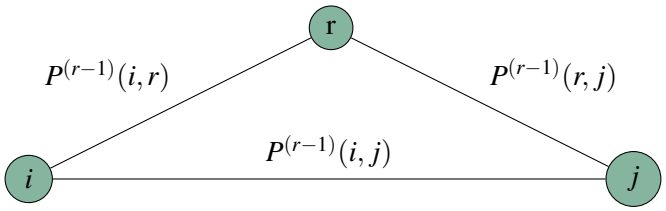
So how do we compute $P^{(n)}$? Let's start with an easier question. How do we compute $P^{(0)}$? The graph's adjacency matrix is $P^{(0)}$. An entry in this matrix is 1 only if there is a path from i to j that doesn't pass through **any** intermediate vertex (and therefore doesn't pass through any intermediate vertex numbered higher than 0). Perfect! That's just what $P^{(0)}$ should be.

Now let's think about how to compete $P^{(1)}$. If there is a path from i to j that passes through an intermediate vertex numbered no higher than 1, then it either goes through node 1, or it doesn't:



therefore, $P^{(1)}(i,j) = \max(P^{(0)}(i,1) \cdot P^{(0)}(1,j), P^{(0)}(i,j))$. Knowing this, it's easy now to construct $P^{(1)}$ from $P^{(0)} = A$. We just perform this computation once to get each entry of $P^{(1)}$.

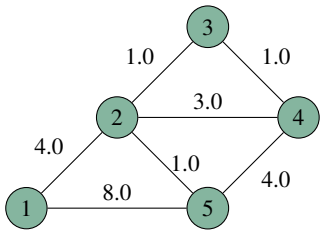
Now let's consider the general case. Suppose we have computed $P^{(r-1)}$ and now we want to compute $P^{(r)}$. The argument is similar to the one we made above. If there is a path from i to j that passes through a vertex numbered no higher than r, then it either uses vertex r or it doesn't:



We know the number of paths from i to r that don't pass through a node higher than $r-1$, that's $P^{(r-1)}(i,r)$. We know the number of paths from r to j that don't pass through a node higher than $r-1$, that's $P^{(r-1)}(r,j)$, and we know the number of paths from i to j that don't pass through a vertex numbered higher than $r-1$, that's $P^{(r-1)}(i,j)$. Thus, there is a path from i to j that passes through a vertex numbered no higher than r if and only if either both $P^{(r-1)}(i,r)$ and $P^{(r-1)}(r,j)$ are true, or $P^{(r-1)}(i,j)$ is true. Therefore

$$P^{(r)}(i,j) = \max(P^{(r-1)}(i,r) \cdot P^{(r-1)}(r,j), P^{(r-1)}(i,j)).$$

Since we now know how to compute $P^{(r)}$ from $P^{(r-1)}$ for any r, it is easy to compute $P^{(n)}$. We start with $P^{(0)} = A$, then from that we compute $P^{(1)}$. From $P^{(1)}$ we compute $P^{(2)}$, and so on, all the way up to $P^{(n)}$. Then $P^{(n)}(i,j)$ tells us if there is a path from i to j. We can even do this without using any intermediate storage! The pseudocode for Warshall's algorithm follows.



	1	2	3	4	5
1	0	4	∞	∞	8
2	4	0	1	3	1
3	∞	1	0	1	∞
4	∞	3	1	0	4
5	8	1	∞	4	0

Figure 18.1: A undirected weighted graph and its weight matrix.

```
// Warshall's algorithm for determining path existence.
Algorithm pathExistenceWarshall(A)
A - adjacency matrix of a graph

P = A
for r=1 to n
    for i = 1 to n
        for j = 1 to n
            P(i,j) = max( P(i,r) * P(r,j), P(i,j) )

return P
```

Note that Warhsall's algorithm for path existence is actually an "all-pairs" algorithms in that it determines whether a path exists between all pairs of nodes.

Optional Reading: Transitive Closure

This section is optional reading and will not appear on any examinations.

The transitive closure of a graph G is the graph that results if you add to G a direct edge between every pair of nodes in G which are connected by a path of any length. This is what Warshall's algorithm is actually computing! If you treat $P^{(n)}(i,j)$ as an adjacency matrix, then it represents the transitive closure of the graph represented by A.

18.2 Path Algorithms for Weighted Graphs

A *weighted graph* is a graph in which each edge has an associated real number called a *weight*. This weight represents the cost of traversing the edge in the context of whatever is being modelled by the graph, for example, the distance in kilometres of the road between two cities, the data transmission time between two nodes in a network, the cost of airfare for a direct flight between two cities, etc..

An adjacency matrix is not sufficient to represent a weighted graph. Instead we use a *weight matrix*, W, in which each entry $w(i,j)$ is the weight of the edge from i to j. In general, a weight can be any real, finite value, so the lack of an edge has to be indicated with some special value, such as $+\infty$. One could also use an adjacency list representation for a weighted graph where each node in the adjacency list stores both the index of the adjacent node and the weight of the edge. Figure 18.1 shows an undirected weighted graph and its corresponding weight matrix. A directed weighted graph is represented in the same way; the only difference is that the weight matrix is not symmetric.

18.2.1 Shortest Paths in a Weighted Graph

In a weighted graph, a shortest path between two nodes (if a path exists at all) is not the path with the fewest edges but rather the path from i to j with the smallest sum of weights along the edges of the path. For example, in Figure 18.1, the shortest path from node 1 to node 4 is 1, 2, 3, 4, even though the path 1, 2, 4 has fewer edges. This is because the sum of the weights of the edges along the former path, $(1,2)$, $(2,3)$, and $(3,4)$ is 6, and the sum of the edges $(1,2)$, $(2,4)$ along the latter path is 7.

18.2.2 All-pairs Shortest Paths — Floyd's Algorithm

Floyd's algorithm (1962) determines the length of the shortest paths between all pairs of nodes in a weighted graph. The derivation of the algorithm is very similar to that of Warshall's algorithm (and with good reason). Let $D^{(k)}$ be a matrix in which entry $d^{(k)}(i,j)$ is the length of the shortest path from node i to node j that does not pass through any other nodes numbered higher than k.

$D^{(0)}$ is easy to compute, it is just W, because entry $w(i,j)$ of W is the length of the shortest path from i to j that doesn't pass through any other intermediate nodes. The generalization is very similar to that of Warshall's algorithm. If we know D^{r-1} then $d^{(r)}(i,j)$ is equal to the smaller of:

- the length of the shortest path from i to k + the length of the shortest path from k to j (i.e. $d^{(r-1)}(i,r)+d^{(r-1)}(r,j)$); and
- the length of the shortest path from i to j that doesn't go through r (i.e. $d^{(r-1)}(i,j)$).

In other words:

$$d^{(r)}(i,j) = \min(d^{(r-1)}(i,r)+d^{(r-1)}(r,j), d^{(r-1)}(i,j)).$$

For the same reasons as for Warshall's algorithm, it is sufficient to compute $D^{(n)}$ to find the shortest paths, otherwise we are just repeating vertices on cycles (which will then no longer be paths). Entry $d^{(n)}(i,j)$ of $D^{(n)}$ contains the length of the shortest path from i to j in the graph. To find $D^{(n)}$ we start with $D^{(0)} = W$, use that to compute $D^{(1)}$, use $D^{(1)}$ to compute $D^{(2)}$, and so on up to $D^{(n)}$. Like Warshall's algorithm, we can do this without any additional storage. Pseudocode for Floyd's algorithm follows.

```
// Floyd's algorithm for all-pairs shortest paths in a weighted graph.
Algorithm shortestPathFloyd(W)
W - weight matrix of a weighted graph

D = W
for r=1 to n
    for i = 1 to n
        for j = 1 to n
            D(i,j) = min( D(i,j), D(i,r) + D(r,j) )

return P
```

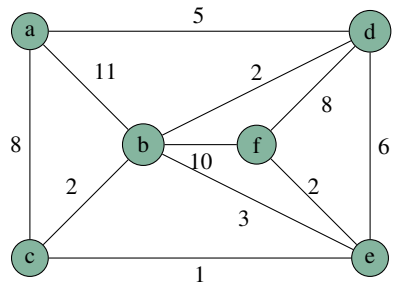
There's a reason that Floyd's algorithm and its derivation looks so similar to Warshall's algorithm. It's because they're the **same** algorithm! Warshall discovered the algorithm for unweighted graphs independently from Floyd who invented it for weighted graphs. Warshall's algorithm is really just a special case of Floyd's algorithm where the weight of every edge is 1. For this reason, what we have presented as "Floyd's algorithm" is often referred to as the Floyd-Warshall algorithm.

Note that Floyd's algorithm only determines the lengths of the shortest paths. With some modification and an additional $n \times n$ matrix, the actual shortest paths can be reconstructed. This is beyond the scope of our course, but those interested might wish to consult the Wikipedia article on the Floyd-Warshall algorithm.

18.2.3 Single-source Shortest Paths with Non-Negative Weights — Dijkstra's Algorithm

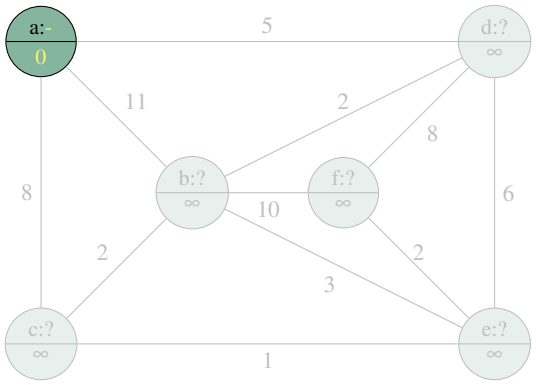
Single-source shortest paths is the problem of finding the shortest path from a single start node to each other node in the graph. Note that Floyd's algorithm solves this problem for every possible start vertex. But if we only need to know the shortest paths from a single start vertex, it is better to solve the single-source shortest paths problem because it requires a lot less time. The best algorithm for solving single-source shortest paths is Dijkstra's algorithm, published by Edsgar Dijkstra in 1959. The most efficient implementation of Dijkstra's algorithm was published in 1984 by Fredman and Tarjan (teaser: it uses a priority queue!).

Before we look at the pseudocode algorithm for Dijkstra's algorithm, let's do an example. Consider the following graph $G = (V, E)$ where V is the set of nodes $\{a, b, c, d, e, f\}$:



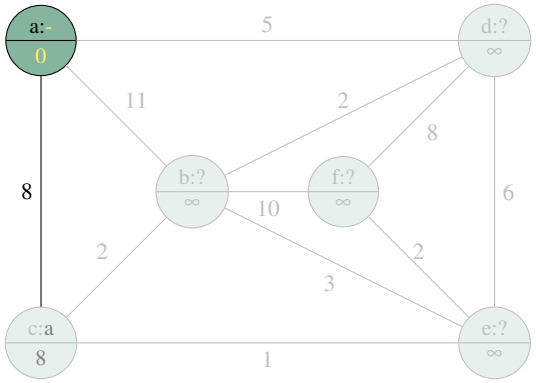
Suppose we want to know the shortest paths from node a to every other node. Initially the start node is the current node and is marked as "visited," and all other nodes are marked "unvisited." For each node i, we will keep track of the length of the shortest path found so far from a to i (this will be shown by a yellow number in the lower half of the node) and the node immediately prior to i on the shortest path to i found so far (this will be shown by a yellow node letter in the upper half of the node). We will call the former the *tentative distance* and the latter the *predecessor node*.

Initially, we only know this information for the start vertex. We know that the shortest path from a to itself is of length 0, and that there is no previous vertex on the path from a to a. Unvisited nodes will be shown lightened, so our initial situation looks like this:

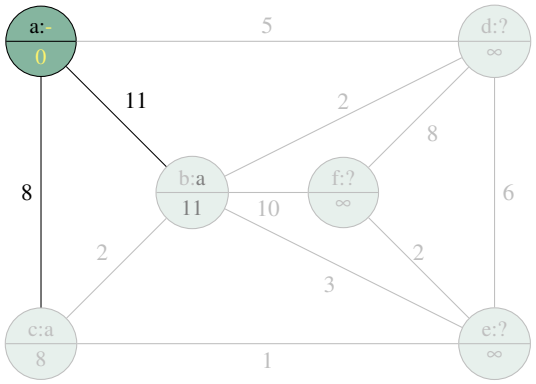


For nodes other than the start node a, the initial tentative distance is ∞ because we don't know any paths from a to those nodes yet, and there are no predecessor nodes for the same reason (denoted by a ?). The algorithm proceeds by processing the current node, which, at present, is a. To process the current node, we always will look at all unvisited nodes that are adjacent to the current node. Let's consider node c, unvisited, and adjacent to a. We can now record that we know a path from a to c

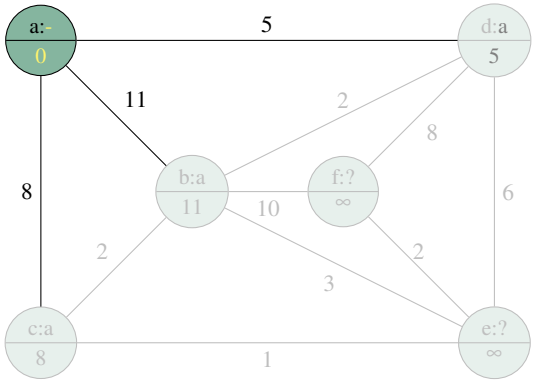
which has a cost of 8, and on which the predecessor of c is a. So we can record that, now showing edges that belong to (tentative) shortest paths in black:



The next unvisited vertex adjacent to a is b. We now know that there is a path of length 11 from a to b, on which b's predecessor is a. So we can record that too:

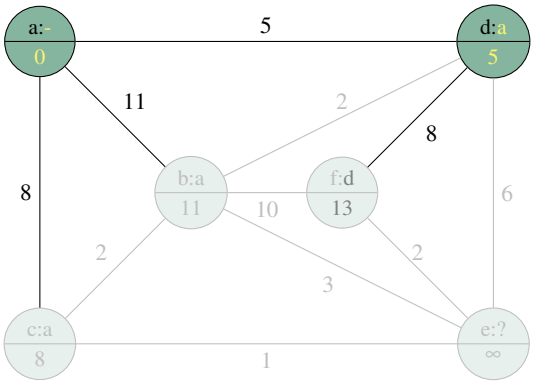


The last unvisited vertex adjacent to a is d, so we record the fact that there is a path from a to d of length 5, on which the predecessor of d is a:

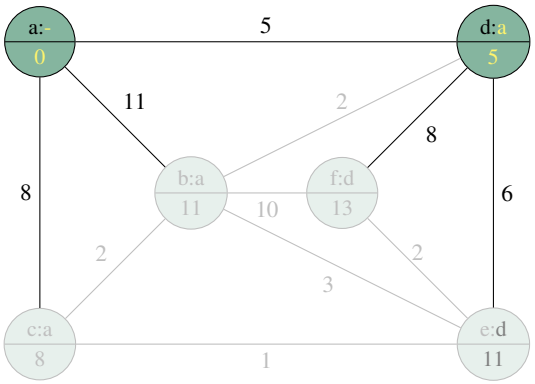


This concludes the processing of node a. Once a vertex is visited and processed, it is never visited and processed again. The big question now is, which unvisited node becomes the next current node which we then process? The answer is: we always choose the unvisited node with the smallest tentative distance. In the above graph, this is node d, with a tentative distance of 5. So now we visit d and start processing its adjacent, unvisited vertices.

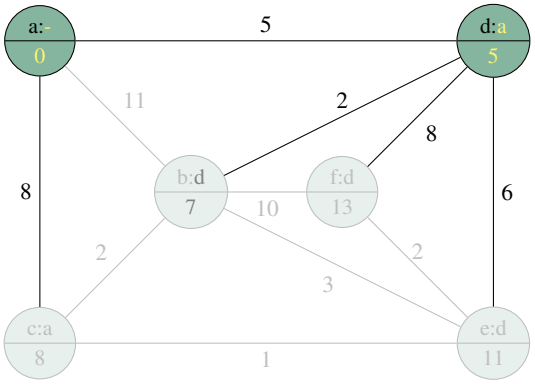
The first unvisited vertex adjacent to d is f. Now we know that there is a path from a to f with cost equal to the tentative distance to d (5), plus the cost of the edge from d to f (8), or 13, and on which the predecessor of f is d. So we record that:



The next unvisited node adjacent to d is e. We now know that there is a path from a to e with cost equal to the tentative distance to d (5) plus the cost of the edge from d to e (6), or 11. So we record that:

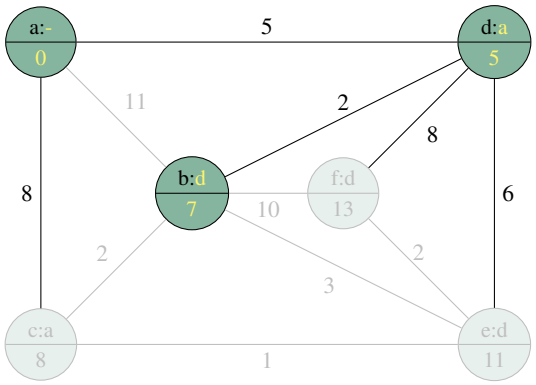


The last unvisited node adjacent to d is b. Now, notice that the contents of node b tell us that we already know there is a path from a to b with a cost of 11, and on which the predecessor of b is a. But what is the cost of the path from a to b that passed through our current node d? It's the cost of the tentative distance to d (5) plus the cost of the edge from d to b (2), or 7! We've found a shorter path to b than we previously knew! If this happens, we update the data in b with the details of the shorter path, in this case, that there is a path from a to b with cost 7, on which the predecessor of b is d:

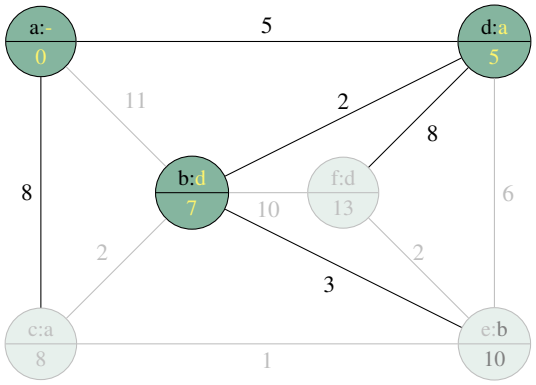


Now we are finished processing node d. To find the next current node, we choose the unvisited node with the smallest tentative distance; this is now node b. So we mark b as visited, and process its unvisited adjacent vertices.

The first unvisited vertex adjacent to b is c. We have just discovered a new path to c which goes via b, whose cost is the tentative distance to b (7) plus the cost of the edge from b to c (2) or 9. This cost of this path is **not** shorter than the tentative distance stored in c, so we know that we already know a shorter path to c than the current path to c, so we do not update anything (the only difference now is that b is marked as visited):

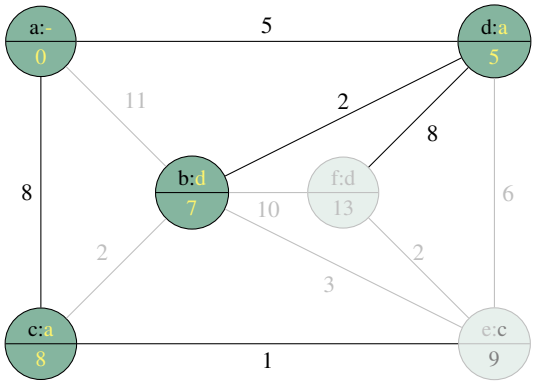


Now we consider f which is adjacent to b and unvisited. The cost of the path from a to f is the tentative distance to b (7) plus the cost of the edge from b to f (10) or 17. This is **not** shorter than the existing tentative distance to f, so again, nothing gets changed because we didn't find a shorter path to f than the one we already knew. Finally, we consider e which is adjacent to f and unvisited. The existing tentative distance in e is 11, which means that we have found a shorter path to e. It has cost equal to the tentative distance to b (7) plus the cost of the edge from b to e (3) or 10. So we record in node e that we know a path from a to e of cost 10, on which the predecessor of e is b:



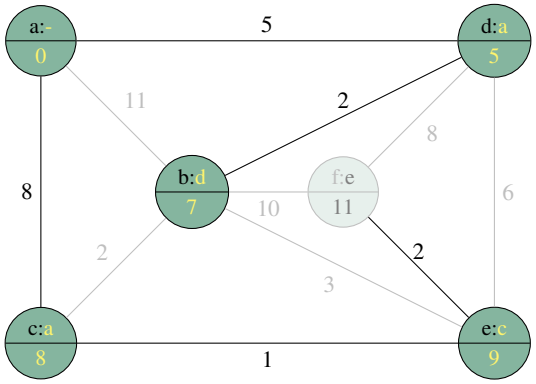
We are now finished processing node b.

The next current node will be c because it has the smallest tentative distance of the unvisited nodes. So we mark c as visited, and the consider its only adjacent unvisited node, e. The path from a to e via c has a cost equal to the tentative distance to c (8) plus the cost of the edge from c to e (1) or 9. We have now found a shorter path to e than we already knew because the contents of e are a tentative distance of 10, via the path from node b. So we update the contents of e to reflect this new, shorter path via c:



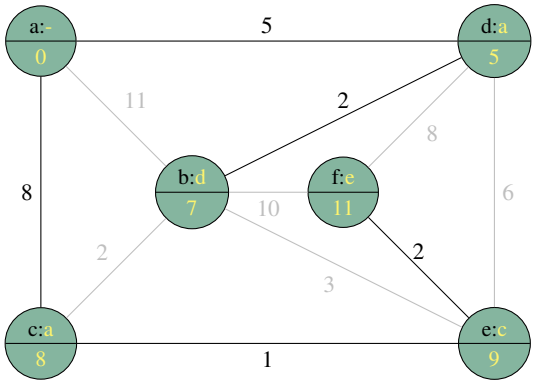
Now we are done processing node c.

The next current node will be e because it has the smallest tentative distance (9) from among the remaining unvisited nodes. So we mark e as visited and consider its only adjacent unvisited node, f. The cost of the path from a to f via e is the tentative distance to e (9) plus the cost of the edge from e to f (2), or 11. Thus, the path from a to f via e is shorter than the path from a to f via d that we had previously recorded, so we replace the contents of node f to reflect that:



We are now finished processing node c.

The next current node will be f because it is the only unvisited vertex remaining, and therefore must have the lowest tentative distance of the remaining unvisited nodes. So we mark f as visited. Since there are no unvisited nodes adjacent to f, we are immediately finished processing node f. Since there are no remaining unvisited nodes, we are done, and the final result looks like this:



At the conclusion of this process, the tentative distances are now final distances—these **are** the costs of the shortest paths to each node. The shortest paths themselves can be found using the predecessor nodes recorded in each node. Note that the edges that participate in shortest paths (the black edges) form a tree. The actual shortest path from a node can be reconstructed by following the predecessor nodes. For example, if we want the shortest path from a to f, we start at f. Its predecessor is e. e's predecessor is c, c's predecessor is a, and a has no predecessor. Concatenating these in reverse order gives us the shortest path from a to f: a, c, e, f.

So the key points to remember are:

- The current vertex v is processed by considering each unvisited vertex u adjacent to v, and checking whether the path from a to u via v is shorter than the current tentative distance from a to u. If it is, update u's tentative distance and predecessor accordingly.
- When finished processing a vertex, the next vertex to process is always the unvisited vertex with the smallest tentative distance.

Below is the pseudocode for Dijkstra's algorithm. Dijkstra's algorithm works regardless of whether the graph is directed or undirected, weighted or unweighted. An unweighted graph is treated like a weighted graph where all of the weights are 1. In a directed graph, a node u is only adjacent to v if (v, u) is an edge. However, Dijkstra's algorithm does **not** work at all if the graph contains any negative weights. Try to re-do the previous example on paper while tracing through the pseudocode:

```
Algorithm dijkstra(G, s)
G is a weighted graph with non-negative weights.
s is the start vertex.

Let V be the set of vertices in G.

For each v in V
    v.tentativeDistance = infinity
    v.visited = false
    v.predecessorNode = null

s.tentativeDistance = 0

while there is an unvisited vertex
    cur = the unvisited vertex with the smallest tentative distance.
    cur.visited = true

    // update tentative distances for adjacent vertices if needed
    // note that w(i,j) is the cost of the edge from $i$ to $j$.
    For each z adjacent to cur
        if (z is unvisited and z.tentativeDistance >
                                cur.tentativeDistance + w(cur,z) )
            z.tentativeDistance = cur.tentativeDistance + w(cur,z)
            z.predecessorNode = cur
```

19 — Sorting Algorithms

Learning Objectives

After studying this chapter a student should be able to:

- demonstrate an understanding of how the presented sorting algorithms work, including:
  - Insertion sort
  - Selection sort
  - Bubble sort
  - Quick sort
  - Merge sort
  - Tree sort
  - Heap sort

  (you should be able to simulate the steps of each algorithm by drawing pictures on paper given an input array); and
- state the best-case, worst-case, and where appropriate, average-case time complexity of each of the above sorts.

19.1 Review

This section provides a quick review of the sorts covered in CMPT 111. Some portions of this section are inspired by, and some portions are blatantly stolen from Daniel Neilson and Michael Horsch, *Introduction to Computer Science and Programming: Course readings for CMPT 111 and CMPT 116* (third edition), unpublished, September 2013. This wanton act was done with permission of the authors. After we quickly review insertion sort, selection sort, and bubble sort, we will move on to discussing other, more efficient algorithms.

19.1.1 Insertion Sort

The insertion sort repeatedly assumes that the first k elements of the array (starting at $k = 1$) have already been sorted. Insertion sort chooses the next unsorted value in the array (at offset k) and looks

for the right place to put it in the portion of the array that is already sorted. This increases the size of the sorted portion by one. The algorithm stops when the entire array has been sorted (when $k = n$). The sort derives its name from the repeated insertion of a value into the sorted portion of the array.

```
Algorithm insertionSort(data)
data - array to be sorted

for k = 1 to data.length-1
    // Place the value at data[k] into its proper place in the subarray
    // from offsets 0 through k
    savedValue = data[k]
    int position = k
    while position > 0 and data[position-1] > savedValue
        data[position = data[position-1]
        position = position - 1
```

In the best case, the array is already sorted; the k-th element is always already in place, so the while loop condition is always false. The for loop executes $n-1$ times, and each iteration executes 3 statements. Thus, in the best case, insertion sort is $O(n)$ where $n = data.length$.

In the worst case, the array is in reverse order, and the k-th element needs to be moved to the beginning of the sorted sub-array every time. That means that the while loop condition is always true until `position` = 0, which means each time through the for loop, the while loop condition is true k times for a total of $3k+1$ statements. On iteration k, the for loop therefore requires $2+3k+1$ statements. Over all values of k this becomes:

$$\sum_{k=1}^{n-1}(3+3k) = 3\frac{n(n+1)}{2} + 3(n-1) = \frac{3}{2}(n^2+n) + 3(n-1)$$

which is $O(n^2)$.

An interesting property of insertion sort is that if each element in the array starts no more than k positions from its sorted position, then the time complexity of the sort is only $k \cdot O(n)$ or $O(n)$. This makes insertion sort very efficient even if the input array is not quite perfectly sorted, but rather **almost** sorted. Generally speaking, the closer to being sorted an array is, the more closely the time required by insertion sort approaches $O(n)$.

19.1.2 Selection Sort

Selection sort works very much the way humans would sort an array. Suppose we had a small array of values, and we wanted to sort from low to high values (ascending/increasing order). We select the smallest value from the whole array, and move it to the beginning of the array. Then we'd find the next smallest value, and move it to the 2nd position. We'd keep doing this until the whole array is sorted. The name is derived from the fact that we are selecting the value we need "next" from the unsorted portion of the array.

Starting with $k = 0$, the selection sort repeats the following three steps as long as k is less than the number of elements in the array:

1. Let s be the offset of the smallest value from elements k through to the end of the array.
2. Swap the values at offsets s and k.
3. Increment k by one.

This algorithm, as outlined, will sort a list into increasing order. If you need to sort the array into descending/decreasing order instead, then you would search for the largest value in step 1.

```
Algorithm selectionSort(data)
data - array to be sorted

for k=0 to data.length-1
    // Perform one sweep:
    // Place the smallest value from offsets [k..size-1] into offset k

    // Find position of smallest value in [k .. size-1]
    position = k;
    for i=k+1 to data.length-1
        if data[i] < data[position]
            position = i;

    // swap values at: k & position
    if position != k
      int temp = data[k];
      data[k] = data[position];
      data[position] = temp;
```

For selection sort, the best case and the worst case are the same. No matter how many swaps occur, we must still search the entire subarray from offsets k to $data.length-1$ for the smallest element. This means that the inner for loop makes $n-k-1$ array element comparisons. Another way to say this is that the if-statement in the inner loop is the active operation and the active operation executes $n-k-1$ times each time the for loop is executed. This has to happen for every value of k from 0 to $n-1$. thus, the total number of executions of the active operation is:

$$\sum_{k=0}^{n-1}(n-k-1)=\sum_{k=0}^{n-1}k=\frac{n(n-1)}{2}=n^2/2+n/2$$

which is $O(n^2)$. Since the best and the worst cases are the same, we can say that selection sort is $\Theta(n^2)$.

19.1.3 Bubble Sort

You can easily tell if an array is unsorted: For example, the numbers $1,2,4,3,5$ are not sorted, and the values $4,3$ are adjacent, but not in the correct order. Bubble sort is a technique that tries to correct this kind of local inconsistency by swapping adjacent values when they are out of order. In the example, one swap will result in an array that is sorted, but in general, things won't be that easy. Consider the following example: $2,3,5,1,4$. Again, only one pair of adjacent numbers (5 and 1) is not in the correct order, but swapping them does not result in a sorted list.

To ensure that we never make the mistake of saying that an array is sorted when it is not, we have to make sure that we swap every pair of elements that could possibly be out of order. The algorithm for Bubble sort sweeps across the array multiple times, and in every sweep, we check if every pair of adjacent values is in the right order. If they are not, we swap them.

The effect of bubble sort is that, on every sweep, bigger values move one way, and smaller values move the other way. This is similar to the way that bubbles move through a liquid. The bubbles, being lighter than the liquid, move up, and the heavier liquid moves down.

Starting with $k=0$, the algorithm for a bubble sort will sort an array of size n by repeating the following up to n times: Starting at the front of the array, look at each adjacent pair of values in the array; if a pair is out-of-order, then swap the values. If, during a sweep, no swaps occurred, then the array is sorted, and the algorithm stops.

```
Algorithm bubbleSort(data)
data - the array to be sorted
for k = 0 to data.length-1
    // Perform one sweep
    // Go through the entire array, in order, swapping adjacent values
    // if they are out of order.  Note: we start at array offset 1 to avoid
    // out-of-range array access.

    madeSwap = false

    for i = 1 to data.length-1

        // Compare the values at offsets i and (i-1), and swap if they
        // are out of order
        if data[i-1] > data[i]
            t = data[i-1];
            data[i-1] = data[i];
            data[i] = t;
            madeSwap = true

        if !madeSwap
            return
```

In the best case, the array is already sorted. The inner loop will execute $n-1$ times, but no swaps will be made, so `madeSwap` will be false, and the algorithm will return after only one sweep. Thus, in the best case, bubble sort is $O(n)$ where $n = data.length$.

In the worst case, the array is in reverse order. The element at the end of the array will be the smallest element. With each sweep, this smallest element will be moved one offset closer to the beginning of the array where it belongs. This means that `madeSwap` will always be true at the end of a sweep, and the outer loop is not halted early. The inner loop always executed $n-2$ times. The outer loop will execute n times. If we choose the first if-statement of the inner loop as the active operation, we will see that it will execute $n(n-2) = n^2 - 2n$ times, which is $O(n^2)$.

19.1.4 Efficiency Summary

Here's a table showing the relative efficiencies of sorting algorithms when sorting an array of n elements. We'll extend this table as we learn about more sorting algorithms.

Algorithm	Worst-case	Best-case
Bubble sort	$O(n^2)$ (array in reverse order)	$O(n)$ (array already sorted)
Selection Sort	$O(n^2)$ (always performs same number of comparisons)	$O(n^2)$
Insertion Sort	$O(n^2)$ (array in reverse order)	$O(n)$ (array already sorted)

19.2 Sorting Terminology

In this section we define some terminology related to sorts.

A sorting algorithm is *in-place* if the output is stored in the same array as the input, and the algorithm does not use any more than a constant amount of space in addition to the space used by the input array. The bubble, insertion, and selection sorts are all in-place.

A sorting algorithm is *stable* if elements with equal keys stay in the same order relative to each other. The bubble, insertion, and selection sorts are all stable.

19.3 Divide-and-conquer Sorts

There are two very efficient sorts that work on the principle of *divide-and-conquer*. The *divide-and-conquer* approach to problem solving is this:

- **Divide:** Divide the problem to be solved into two smaller sub-problems.
- **Recurse:** Recursively try to solve the sub-problems (possibly by dividing them into even smaller problems). When we eventually reach a level where the sub-problem is small enough, solve it quickly.
- **Conquer:** Take the solutions of the subproblems and combine them into a solution of the original problem.

The divide-and-conquer approach is not specific to solving the problem of sorting, rather it is a general problem solving approach.

We will look at two divide-and-conquer sorting algorithms: *merge sort* and *quick sort*. They are introduced in the following sections.

19.3.1 Merge Sort

Let's suppose we are sorting a sequence of elements S containing n elements. The merge sort algorithm uses the following divide-and-conquer approach:

- **Divide:** If S has zero or one elements, return S immediately, it is already sorted. Otherwise, divide S into S_1 and S_2 such that S_1 contains the first $\frac{n}{2}$ elements of S and S_2 contains the remaining $\frac{n}{2}$ (rounding up and down respectively).
- **Recurse:** Recursively sort the sequences S_1 and S_2.
- **Conquer:** Obtain a sorted version of S by merging the sorted sequences S_1 and S_2.

Observe that the "divide" step is trivial — we just chop the array in half and recursively sort each half. All of the work is done in the "conquer" step. Let's turn this into pseudocode.

```
Algorithm mergeSort(S)
S - array of elements to be sorted

// Divide
S1 = first half of S
S2 = second half of S

// Recurse
S1 = mergeSort(S1)
S2 = mergeSort(S2)

// Conquer!!!
S = merge(S1, S2)
```

Now it should be easy to see that everything hinges on what is going on in the `merge` operation of the "conquer" step. Intuitively, if we have sorted sequences `S1` and `S2` we can "shuffle" them together so that we obtain a sorted version of the original sequence `S`. First, we position a cursor at the start of `S1` and `S2`, respectively. Since `S1` and `S2` are sorted after the recursive call returns, we know that the first element of the sorted `S` has to be either the current element of `S1` or the current element of `S2`, whichever is smaller. We append that element to `S` (which is initially empty) and advance the cursor of that element's sequence. We then do this again, and again, until the cursor of one of the sequences is in the "after" position. Then we just append the remainder of the other list to `S`.

If we are clever, we can do merge sort in-place. We can represent a subarray of an array, `inVec`, to be sorted using three offsets. The first offset, `start1st` is the offset of the start of the first half of the subarray from the divide step. The second offset, `start2nd` is the offset of the start of the second half of the subarray from the divide step, and the third offset, `end2nd` is the offset of the end of the second half of the subarray from the divide step. Suppose that we have recursively sorted the subarrays `inVec[start1st...start2nd-1]` and `inVec[start2nd...end2nd]`. The following java method shows how we can merge these subarrays into a single sorted subarray between offsets `start1st` and `end2nd`. The merge function stores the merged sequence from subarrays `inVec[start1st...start2nd-1]` and `inVec[start2nd...end2nd]` in the temporary array `temp`, and then copies the merged sequence back into `inVec[start1st...end2nd]`.

```
/**     Merge start1st through start2nd-1 with start2nd through end2nd in
        array inVec using temp as a temporary array */
private void merge(int[] inVec, int[] temp, int start1st,
                   int start2nd, int end2nd)
{
    int cur1 = start1st;  // index of the current item in first half
    int cur2 = start2nd;  // index of the current item in first half

    int cur3 = 0;   // index of the next location of temp
    // While neither subarray is empty...
    while ((cur1 < start2nd) && (cur2 <= end2nd)) {
        // Copy the smaller item from the two sequences to temp
        if (inVec[cur1] <= inVec[cur2]) {
            // Note: increments happen after assignment
            temp[cur3++] = inVec[cur1++];
        }
        else {
            temp[cur3++] = inVec[cur2++];
        }

    while (cur1 < start2nd) {
        // copy remainder (if any) of first subarray to temp
        temp[cur3++] = inVec[cur1++];
    }

    while (cur2 <= end2nd) {
        // copy remainder (if any) of second subarray to temp
        temp[cur3++] = inVec[cur2++];
    }

    // Copy the temp array back to inVec[start1st...end2nd]
    cur1 = start1st;
    cur3 = 0;
    while (cur1 <= end2nd) {
        // copy items from temp back into inVec
        inVec[cur1++] = temp[cur3++];
    }
}
```

This design allows us to specify subsequences S_1 and S_2 of S without making copies of the items in the input array except for the temporary storage used in the merge operation.

Once we have the merge operation, the rest of merge sort is a short, recursive algorithm.

```
/** Sort array keys using the recursive merge sort approach */
public void mergeSortRecursive(int[ ] keys)
{
    int[ ] temp = new int[keys.length];
    // merge sort the entire array
    // (the subarray from offset 0 to offset keys.length-1)
    mergeSortHelper(keys, temp, 0, keys.length - 1);
}


/** A helper method to perform a recursive merge sort on the part of the
    array between start and finish */
private void mergeSortHelper(int[ ] inVec, int[ ] temp, int start, int finish)
{
    int size = finish - start + 1; // number of items in current sequence
    if (size > 1)
    {
        // find position of middle item of the subarray (divide step!)
        int middle = (start + finish) / 2;

        // sort first half subarray (recursive step!)
        mergeSortHelper(inVec, temp, start, middle);

        // sort the second half subarray (rest of recursive step!)
        mergeSortHelper(inVec, temp, middle + 1, finish);

        // merge the two sorted subarrays (conquer step!)
        merge(inVec, temp, start, middle + 1, finish);

        // Now the subarray inVec[start...finish] is sorted.
    }
}
```

Merge sort is stable, and as we have seen, can be implemented in-place. We state without argument or proof that merge sort is $\Theta(n \log n)$. We'll look at why this is the case in an in-class discussion.

19.3.2 Quick Sort

Quick sort is another divide-and-conquer approach to sorting in which all of the hard work is done in the "divide" step; contrast this with merge sort where all of the hard work is done in the conquer step. The approach used by quick sort to sort a sequence of elements S is this:

- **Divide**: If S has at least two elements, select some element x from S, called the *pivot*. Remove each element from S and place them in one of three sequences:
  - L (for elements less than x)
  - E (for elements equal to x)
  - G (for elements greater than x)
- **Recurse:** Recursively sort L and G.
- **Conquer:**Put S back together by concatenating the (already sorted) elements of L, E, and G.

Let's express these ideas in pseudocode:

```
Algorithm quickSort(S)
S - array of elements to be sorted

// Divide
pivot = any element of S                // e.g. the last element of S
L = elements in S smaller than pivot    // these elements may be in any order
G = elements in S larger than pivot     // these elements may be in any order
E = elements in S equal to the pivot

// Recurse
quickSort(L)
quickSort(G)

// L and G are now sorted.

// Conquer!!!
S = L + E + G  // (where + represents concatenation)
```

We can already see that the conquer step is easy; we just need to paste together the already-sorted sequences L, E and G because we know that everything in L is smaller than E and that everything in E is smaller than G. Most of the work is done in creating L, E, and G, in the first place. We will call the algorithm for creating L, E and G the *partition* algorithm. With a little bit of cleverness, the partitioning of S can be done in-place such that the first part of S is L, the second part of S is E, and the third part of S is G.

Let's illustrate this with an example. Suppose we decide to always pick the last element in the array S as the pivot. Then we want to arrange the array so that the items smaller than the pivot are left-most in the array, followed by the pivot, followed by the items that are larger than the pivot. But, other than the pivot, we don't care if the remaining elements are in the correct positions. If the array is of length n, then we can do this by starting with two integers l, and r, that refer to offsets in the array, such that l is initialized to 0 and r is initialized to $n-2$. Thus, l and r start at either ends of the array (but not including the pivot at the end of the array). This setup is shown as step 1 of Figure 19.1. After this initialization, we increment l until $S[l]$ is larger than the pivot, and decrement r until $S[r]$ is smaller than the pivot. The result of this is shown in Step 2 of Figure 19.1. At this point we swap $S[l]$ and $S[r]$ because they are on the wrong sides of the array; this is shown in Step 3. We always want smaller elements to be to the left of larger elements. Then we repeat this process until l is larger than r. Step 4 shows the next movements of l and r, and step 5 shows the resulting swap. Step 6 shows that when we move l and r again, we end with $l > r$. This tells us that no more swaps are necessary; all of the elements smaller than the pivot now appear in the array before all the elements larger than the pivot. To complete the partitioning, we swap $S[l]$ and the pivot. Now all the elements smaller than the pivot appear first in the array, then the pivot, then the elements larger than the pivot. These sub-arrays correspond with the sequences L, E and G in the above pseudocode algorithm.

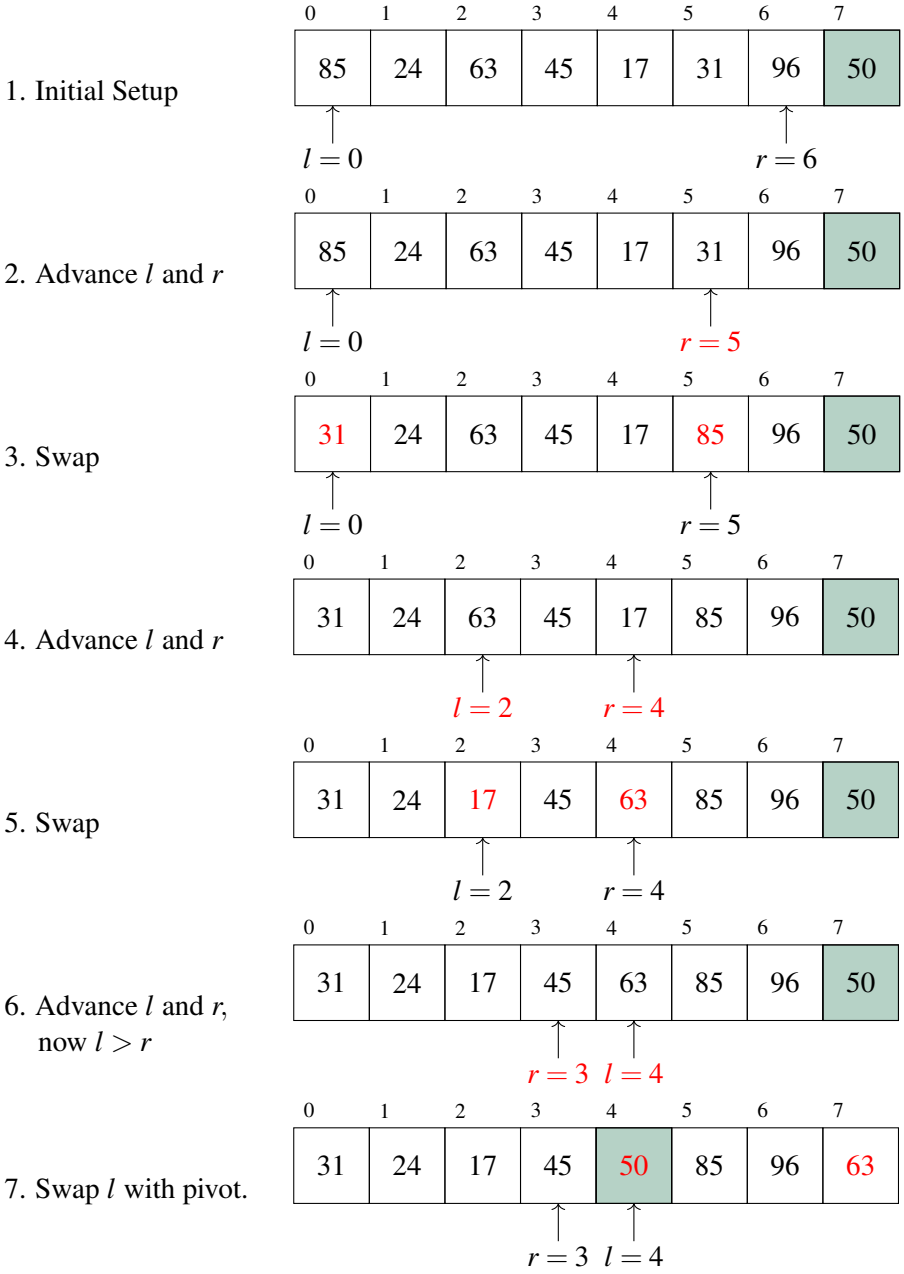


Figure 19.1: Example of partitioning an array (the "divide" step of quick sort) when the pivot is 50. At conclusion of the partitioning, notice that subarray from offsets 0 to 4 contain the elements smaller than the pivot, offset 5 contains the pivot, and offsets 5 to 7 contain the elements larger than the pivot. These subarrays correspond to the sequences L, E, and G, respectively.

The following Java method implements the in-place partitioning of a (sub-)array:

```
/**
 In-place partitioning of the sub-array of array inVec
 bounded by offsets 'start' and 'end'.
 @param inVec: array in which partitioning is to take place
 @param start: starting offset in inVec of the subarray to be partitioned
 @param finish: ending offset in inVec of the subarray to be partitioned
 @return new array offset of the pivot after partitioning
 */
private int partition(int[ ] inVec, int start, int finish)
{
    int partitionItem = inVec[finish]; // item that will be the pivot
    int l = start;  // initial lower offset
    int r = finish - 1; // initial upper offset
    while (l < r)
    {
        l++;
        // move l right, passing "small" items
        while ((l < r) && (inVec[l] <= partitionItem))
            l++;
        r--;
        // move r left, passing "large" items
        while ((l <= r) && (inVec[r] >= partitionItem))
            r--;

        // At this point, r is the offset of a "small" item
        // and l is the offset of a "large" item.

        if (l < r)  // If l and r have *not* yet crossed...
        {
            // swap the large item at offset l with the
            // small item at offset r
            int temp = inVec[l];
            inVec[l] = inVec[r];
            inVec[r] = temp;
        }
    }
    // put item l in the last offset of the subarray, and
    // partitionItem (the pivot) in position l
    inVec[finish] = inVec[l];
    inVec[l] = partitionItem;
    return l;
}
```

Once we have the in-place partitioning algorithm implemented, the rest of quick sort is quite straightforward. An implementation of the quick sort pseudocode, given above, is shown in the following java method:

```
public void quickSort(int[ ] keys)
{
    // Quick-sort the entire array
    quickSortHelper(keys, 0, keys.length - 1);
}

// Quick-sort the subarray of inVec between offsets 'start' and 'finish'
private void quickSortHelper(int[ ] inVec, int start, int finish)
{
    if (start < finish)
    {
        // Divide step
        int pivotOffset = partition(inVec, start, finish);
        // now small items are to the left of pivotOffset,
        // and large items to the right of pivotOffset
        // The pivot item is in the correct final position at pivotOffset!

        // Recursive steps:

        // Recursively sort the subarray inVec[start...pivotOffset-1]
        quickSortHelper(inVec, start, pivotOffset - 1);

        // Recursively sort the subarray inVec[pivotOffset+1, end]
        quickSortHelper(inVec, pivotOffset + 1, finish);

        // Conquer step:  Nothing to do!  The subarray inVec[start...finish]
        // is sorted!  Because we did everything in-place, there is no
        // need to explicitly concatenate anything.
    }
}
```

As we have seen, quick sort may be implemented in place, however, it is **not** stable due to the nature of the `partition` algorithm.

We state, again without proof or argument, that quick sort is $O(n \log n)$ on average, but is $O(n^2)$ in the worst case, where n is the number of elements to be sorted. It is also $O(n \log n)$ in the best case. It may, therefore, surprise you to learn that quick sort is usually a little faster than merge sort in practice. The reason is that, in practice, the worst-case is very rare. We'll examine these issues in more detail in class.

19.4 Tree-based Sorts

In this section we briefly discuss two sorts based on tree data structures.

19.4.1 Tree Sort

Tree sort uses a balanced tree to achieve sorting. The basic idea is to insert all of the elements into the tree, then perform a traversal of the tree that outputs the elements in sorted order. For example, suppose we used an AVL tree. We could insert each element from the array to be sorted into an AVL tree. Then, we could do an in-order traversal of the AVL tree where the visit operation is to append

the element in the current node to the sorted sequence. Each insertion would take $O(\log n)$ time, which would be done n times, resulting in a requirement of $O(n \log n)$ time to build the tree. Add to this the cost of the traversal, which we know from studying traversals is $O(n)$, and we have total time for Tree Sort of $O(n \log n + n)$ or just $O(n \log n)$.

Thus, Tree Sort has a guaranteed performance of $O(n \log n)$ time. However, the constant (which Big-Oh notation ignores) is almost always larger than for merge sort or quick sort. As a result, tree sort is usually not competitive with merge sort or quick sort in practice.

Tree sort is not in-place, but it is stable if elements from the input array are inserted into the tree in their original order. Since the second of two equal elements in the input will always be inserted into an AVL tree to the right of the first, the equal elements will always be visited in the traversal in the same order in which they appeared in the input array.

```
Algorithm treeSort(S)
S - array of elements to be sorted

T = new empty AVL tree (or similar balanced tree)
for each item in S
    T.insert(item)

// do in-order traversal (or other traversal appropriate to chosen tree)
// to extract the elements in sorted order.
S = T.extractInorderSequence();
```

19.4.2 Heap Sort

Heap sort works by converting the input array S into an arrayed heap. In such a heap, the largest element is at the root of the heap. The sorting is achieved by repeatedly moving the largest element into its correct position, and re-heapifying the remainder of the array.

You will recall that when we previously discussed arrayed heaps (in a programming assignment) the root was stored at offset 1 of the array, and then the left and right children of the data at any offset i could be found at offsets $2i$ and $2i+1$ respectively, while the parent of offset i is at offset $i/2$ (integer division). This arrangement minimizes the computations necessary to compute child and parent indices, but is not well-suited to sorting since, in sorting problems, offset 0 normally contains a data item that needs sorting. We can easily modify the design of the heap to accommodate this. We can move the root of the heap to offset 0 of the array, then for an item at offset i, its left child is at offset $2i+1$, its right child is at offset $2i+2$, and its parent is found at $(i-1)/2$ (integer division).

Converting the Array to a Heap

The first task in heap-sort is to turn our input array of n elements into a heap: for each element from right to left, as long as it is smaller than any of its children we swap it with its larger child. If we think about it though, we don't have to start at the right-most element, because we are guaranteed that the right-most $n/2$ elements will be leaf nodes, and therefore have no children with which to swap! So we can start this process at index $i = n/2 - 1$.

Figure 19.2 shows how this process described above works. Step 1 shows the input array to be sorted. Beginning at offset $i = 7/2 = 3$ we check whether the element at offset 3 is smaller than either of its children. Its left (and only) child is at offset $2 \times 3 + 1 = 7$, and contains the item 50, so we swap 45 and 50. Since 45 is now a leaf node, no further swaps can occur. The result of processing offset 3 is shown in step 2 of Figure 19.2. Next we decrement i and process offset 2. The element at offset 2 is 63 and its children are at offsets $2 \times 2 + 1 = 5$ and $2 \times 2 + 2 = 6$ which are the elements 31

and 96 respectively. 63 is smaller than 96, so we swap 63 with 96. 63 is now at offset 6, and is a leaf node, so no further swaps are possible and we are finished processing offset 2. The result after processing offset 2 is shown in Step 3. *i* is decremented again, and is now equal to 1, so we process the item at offset 1 which is 24. The children of 24 are at offsets 3 and 4, which are the elements 50 and 17 respectively. 24 is smaller than 50 so we swap 24 and 50 which leaves item 24 at offset 3. 24 now has one child at offset 7 which is 45. Since 24 is smaller than 45 we swap again and 24 ends up in offset 7. Offset 7 has no children, so no further swaps occur and we are finished processing offset 1. The result of processing offset 1 is shown in Step 4. Finally we decrement *i* to 0 and process item 8 at offset 0. The children of 8 are at offsets 1 and 2, and are currently 50 and 96 respectively. 8 is smaller than both of these, so we swap with the larger child, 96, and item 8 ends up at offset 2. Item 8's new children are offsets 5 and 6, and are 31 and 63, respectively. Again 8 is smaller than both its children, so it is swapped with the larger one, 63, and item 8 ends up at offset 6. Item 8 is now a leaf node, so we are done processing it. The result is shown in Step 5 of Figure 19.2. To convince you that the array now represents a heap, we have drawn the array as a heap at the bottom of Figure 19.2; note that each node's item is larger than that of its children.

The algorithm for converting an array into an arrayed heap, which we call `heapfiy`, follows.

```
Algorithm heapify(data)
data - array of elements to convert to a heap

n = data.length
// We can start at i = n/2 because larger offsets are guaranteed
// to be leaves by the properties of arrayed heaps.
for i = n/2 to 0
     // Make data[i] the root of a valid heap
     // by swapping data[i] with children repeatedly as needed.
     moveDown(data, i, n-1)



Algorithm moveDown( data, first, last )
data - array of elements to be converted to heap
first - offset of element to process
last - offset of last element in the 'data' array

// select the left child of offset 'first'
largest = 2 * first + 1;
while( largest <= last )
    // if a right child exists and its item is bigger than data[largest]...
    if( largest < last  and  data [largest] < data[largest+1] )
        largest++         // select the right child instead

    // If item at offset 'first' is smaller than its larger child
    // it does not have the heap property, so swap it with its larger child.
    if( data[first] < data[largest] )
        swap( data[first], data[largest] )  // swap it
        first = largest           // follow the item down
        largest = 2 * first + 1   // select its new left child
    else
        // cause loop to end -- data[first] is the root of a valid heap
        largest = last + 1
```

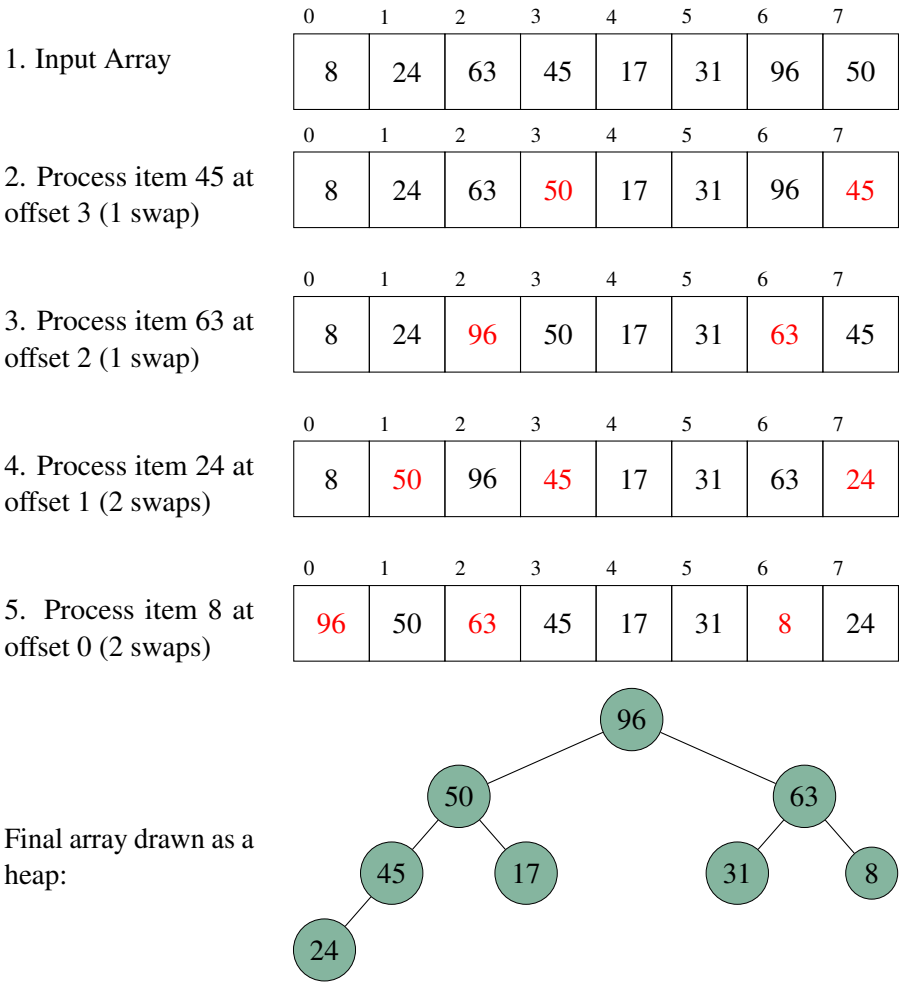


Figure 19.2: Stage 1 of the heap sort algorithm: converting an arbitrary array into an arrayed heap (with the root at offset 0).

Time Complexity of `heapify` Algorithm

`moveDown` performs at most $\log n$ swaps because each time a swap occurs, the swapped item's offset more than doubles. We can only double the item's offset at most $\log n$ times before it becomes a leaf node. Therefore, the loop in `moveDown` executes $O(\log n)$ times. Since a single iteration of this loop requires $O(1)$ time, the complexity of `moveDown` is $O(\log n)$ where $n = data.length$.

In the `heapify` algorithm, the `moveDown()` procedure is called $\lfloor n/2 \rfloor + 1$ times. Therefore the time complexity of heapify is $O(n \log n)$.

Constructing the Sorted Sequence from the Arrayed Heap

The remainder of the heap sort algorithm constructs the final sorted sequence from the heap. The idea is to delete the largest item from the heap, one by one, by swapping the element at the root with the element at the end of the subarray that represents the heap. We will illustrate the process with an example, then present the algorithm.

Step 1 of Figure 19.3 shows the final result of the `heapify` algorithm that we obtained for the input array in Figure 19.2. In normal heap deletion, we just overwrite the root item with the last array element that is part of the heap, and decrease the size of the subarray that represents the heap by 1. What we are going to do here is **swap** the root element with the last element, and decrease the length of the subarray representing the heap by 1! This moves the element from the root of the heap into its correct sorted position because we know it has to be the largest element! We do this repeatedly until the array is sorted. So to start, we swap element 96 with 24, and restore the heap property by swapping 24 with its larger child as long as it is larger than one of its children — this is just the `moveDown` operation again! 24 is larger than its right child, so it swaps with 63. 24's children are now 31 and 8, so it swaps again with 31. The result is shown in Step 2 of Figure 19.3. Now we have a heap with only 7 items, as shown by the green shading. We now know that the root item 63 is the largest of the items remaining in the heap, so we delete it from the heap by swapping it with 8, moving it into correct sorted position, then we call moveDown() on 8 to re-heapify. This causes 8 to swap with its left child 50, then again with its left child 45. The results are shown in Step 3. We then proceed by deleting 50 (swap with 24), then `moveDown` is called again to reheapify and 24 is swapped with its child, 45. The result is shown in step 4. Now 45 is at the root, and it is deleted by swapping it with 17. One swap with the root's right child by `moveDown` restores the heap property. This is shown in step 5. The process continues in this fashion until all elements are in sorted position. The remainder of the example is provided without commentary in Figure 19.4.

The overall heap sort algorithm is given below. It's very short because all of the work is done by the `heapify` and `moveDown` procures.

```
Algorithm heapSort(data)
data - Array of items to be sorted

heapify(data)
i = data.length-1
while( i > 0 )
    // Move the next largest item into sorted position
    swap(data[0], data[i])
    i = i - 1;
    // Re-heapify by moving the new root down.
    moveDown(data, 0, i)
```

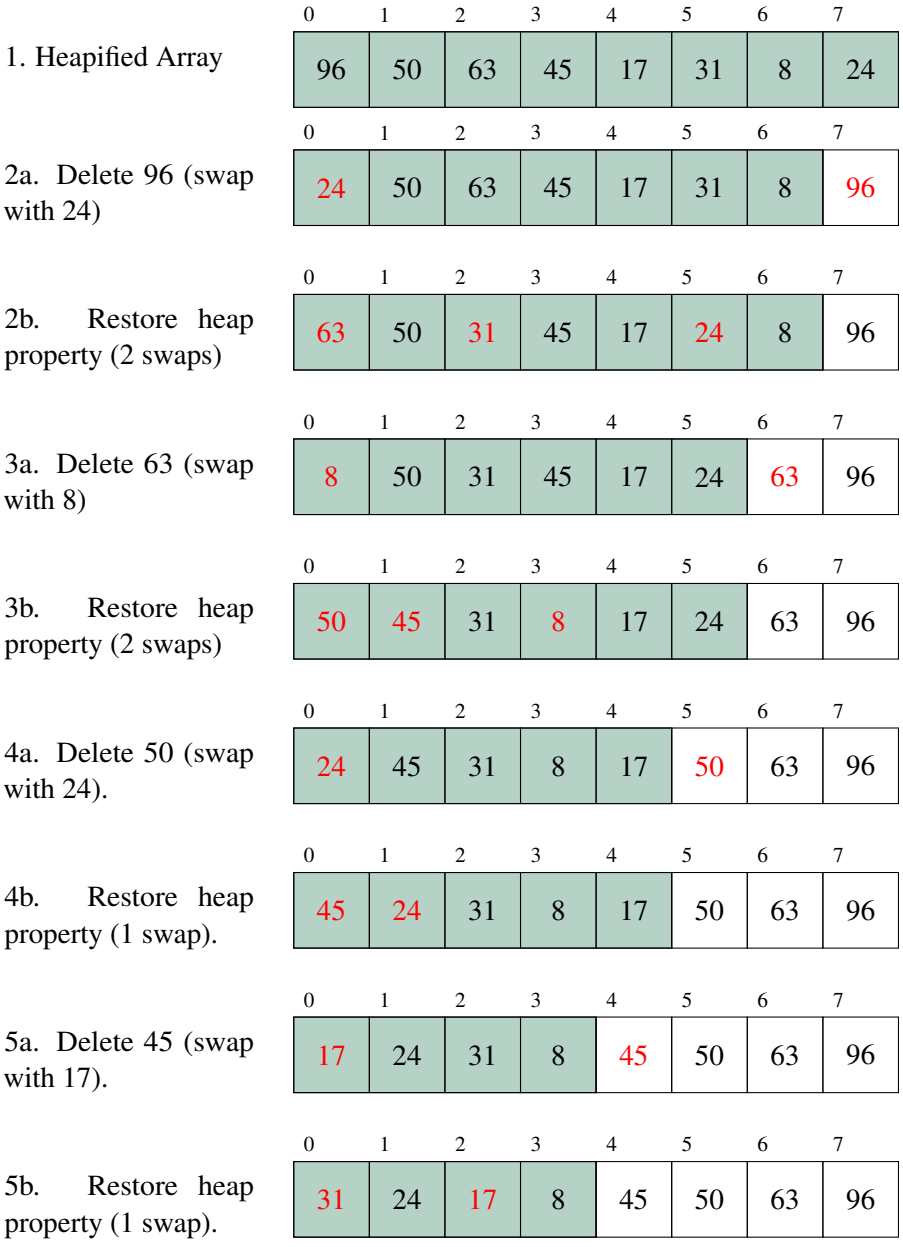
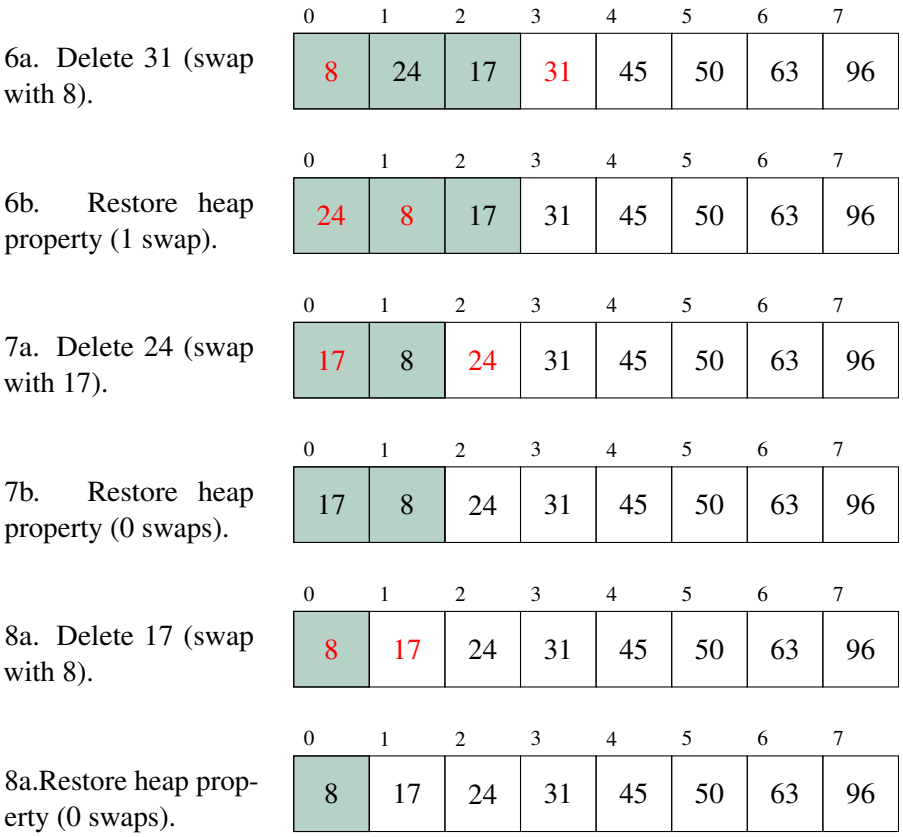


Figure 19.3: Stage 2 of the heap sort algorithm: moving each item into position from largest to smallest, as part of deleting each item from the heap. The array offsets shaded green are part of the heap. This example is continued in Figure 19.4.



The heap now contains only one element, so the array is sorted!

Figure 19.4: Stage 2 of the heap sort algorithm: This figure completes the example in Figure 19.3.

Time Complexity of Heap Sort

Since we already know the time complexities of `heapify` and `moveDown`, the timing analysis of the overall heap sort algorithm is quite straightforward. The first line of the `heapSort` algorithm is `heapify(data);` we know this requires $O(n \log n)$ time. Then we have the while-loop. Each iteration it calls `moveDown`, which we know requires $O(\log i)$ time, which is actually less than $O(\log n)$ time because i is less than n on all iterations. So each loop iteration requires, worst case, $O(\log n)$ time. The loop runs for $n = data.length$ times for $i = data.length$ to $i = 0$. Each of these n loop iterations costs $O(\log n)$, so the while-loop in the `heapSort` algorithm is $O(n \log n)$. The total time for `heapSort` is then the sum of the costs of the call to `heapify` and the while-loop, which is $O(n \log n) + O(n \log n)$ or just $O(n \log n)$.

Heap sort is the "safe," all-purpose sorting algorithm. Generally speaking other sorts are either faster in most situations but have a worse worst-case (e.g. quick sort), have a better worst-case but can't be used in all situations (e.g. radix sort), or have a worse best-case (e.g. bubble sort). Heap sort is not necessarily the best sort for a given situation, but it will always be "pretty good," even in the worst cases. Heap sort is not stable. Our implementation of heap sort is in-place.

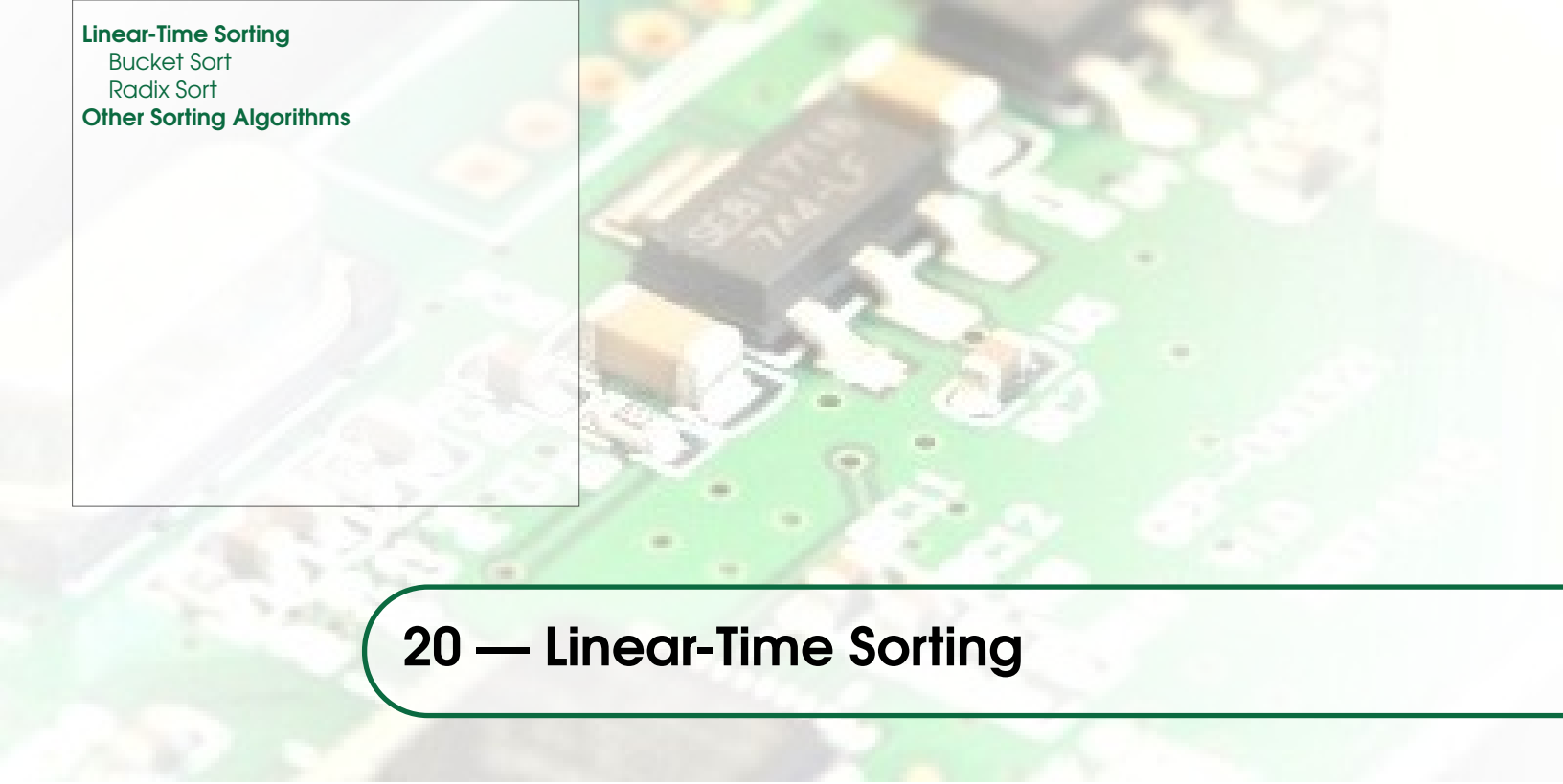
19.5 Efficiency of Sorts — Can we sort faster than $O(n \log n)$ time?

An overview of the efficiency of the sorts discussed in this chapter are shown in Figure 19.5. If you've studied this figure, you've probably noticed that none of the sorts we've covered in this chapter can sort faster than $O(n \log n)$ time. Thus, a natural question on your mind might be: "is it possible to sort in time faster than $O(n \log n)$?" The answer is not as clear-cut as one might think. All of the sorting algorithms we have seen to this point sort elements by comparison. *Sorting by comparison* means determining the relative ordering of elements by comparing entire elements to entire other elements. By now, you're asking: "what other kind of comparison is there?". We'll get to that in a moment. It can be proved, however, that the problem (not algorithm!) of sorting elements by comparison requires $\Omega(n \log n)$ time. Here's a new notation: Big-Ω. It is the opposite of Big-O notation. If a function $f(n)$ is $\Omega(g(n))$ means that $f(n)$ grows *at least as fast* as $g(n)$. Thus, when we say that the problem of sorting by comparison is $\Omega(n \log n)$ it means we have **proven** that $n \log n$ is a lower bound for sorting by comparison, that is, sorting by comparison **cannot be done more quickly** than in time proportional to $n \log n$ by **any** algorithm. Lower bounds (Big-Ω) are very very hard to prove for most problems, so we don't see it very often. But for sorting by comparison, the lower bound is known (you **really** don't want to see the proof though!).

Alas, we cannot sort faster than $O(n \log n)$ if we sort by comparison of elements. It can be proven that this is impossible. But what about other methods of sorting? In the next chapter we will look at some algorithms for sorting that do not directly compare entire elements to each other, rather, they compare pieces of elements to each other. These algorithms can sort a sequence of n elements (of certain limited types) in linear ($O(n)$) time!

Algorithm	Worst-case	Best-case
Bubble sort	$O(n^2)$ (array in reverse order)	$O(n)$ (array already sorted)
Selection Sort	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n^2)$ (array in reverse order)	$O(n)$ (array already sorted)
Quick Sort*	$O(n^2)$ (array already sorted)	$O(n \log n)$ (by luck, pivot is always the median)
Merge Sort	$O(n \log n)$	$O(n \log n)$
Tree Sort**	$O(n \log n)$	$O(n \log n)$
Heap sort	$O(n \log n)$	$O(n \log n)$

Figure 19.5: Best and worst-case time complexities of sorts. *Quick sort is $O(n \log n)$ on average. ** Given times are when using an AVL tree.



20 — Linear-Time Sorting

Learning Objectives

After studying this chapter a student should be able to:

- explain the bucket sort algorithm;
- trace through the MSD and LSD radix sorts for a given input array, showing how each step changes the array; and
- state the time complexities of the bucket sort and the MSD and LSD radix sorts.

20.1 Linear-Time Sorting

At the end of the previous chapter, we noted that sorting by comparison of elements is $\Omega(n \log n)$. That is, there is no sorting-by-comparison algorithm that can sort a sequence of elements faster than time proportional to $n \log n$ where n is the number of elements to be sorted.

In this chapter, we will see one sort that never compares the keys to be sorted to each other at all! We will also see a different approach to sorting in which comparisons take place not between entire keys, but between the digits or characters of a key. This is similar to how trie trees consider the individual digits of a key in contrast with other trees.

20.1.1 Bucket Sort

The first algorithm we will consider is called *bucket sort*. Let's suppose we have a sequence S of n items whose keys are integers between 0 and $d-1$. Note that it is quite possible that $n > d$, so there may be elements with identical keys. The idea of bucket sort is to use the keys as indices (offsets) into a bucket array B.

Each offset of B represents a bucket. For each key k to be sorted, we place it in the k-th bucket, that is, in bucket $B[k]$.[1] Then we can remove the items in each bucket in order, starting from $B[0]$ and concatenate them into a sorted sequence. A pseudocode algorithm for this process follows. Observe that bucket sort never compares keys to other keys!

[1] In practice, each offset $B[k]$ of B is a reference to another sequence of items into which we put all items with key k.

```
// Sort sequence S using element keys.  Assume element keys are
// between 0 and d-1.
Algorithm bucketSort(S)
S - sequence to be sorted

let B be an array of d sequences, each initially empty

for each item x in S
    let k be the key of x
    remove x from S and append it to sequence B[k].

// S is now empty

for i = 0 to d - 1
    for each item x in sequence B[i]
    remove x from B[i] and add it to the end of S.
```

Time Complexity of Bucket Sort

The first for loop executes n times. On each iteration, we remove x from the sequence S which requires $O(1)$ time, and we append it to the sequence $B[k]$ which also requires $O(1)$ time. Thus, the loop body is $O(1)$, and it executes $O(n)$ times, so the first loop requires $O(n)$ time.

The second loop removes each item from one of the buckets ($O(1)$ time), and appends it to S ($O(1)$ time). This happens n times, so the second loop is $O(n)$ as well (if $n > d$). In the special case where $n < d$, the algorithm is $O(d)$ (which is actually the worst-case). But in almost every practical application of bucket sort, $n > d$ and the algorithm runs in $O(n)$ time.

Advantages and Limitations of Bucket Sort

Bucket sort is presented above in its most basic form. The advantages and limitations discussed below pertain to this version.

Bucket sort is excellent for sorting large sequences of integer keys that have a small range of values (i.e. when d is small and much less than n). Thus, bucket sort would be suitable for sorting, say, 8-bit and 16-bit signed or unsigned integer keys.[2]

Bucket sort is not suitable for sorting 32-bit integer keys as this would require $d = 2^{32} \approx 4$ billion buckets. The B array would need to have about 4 billion entries, each of which, on a 32-bit architecture, would have to contain a 32-bit (4-byte) reference, causing the B array alone to require 4×4 billion bytes ≈ 16 **gigabytes** of memory! Space complexity is a limiting factor for bucket sort; it requires $O(d)$ space in addition to the space for storing the keys.

Bucket sort is also not suitable for sorting floating-point values because even if the values of the keys are limited to between 0 and d, there are still an infinite number of real values between 0 and d, which would require an infinite number of buckets. That's not going to happen without, perhaps, an infinite improbability drive system.

It is not possible to sort character strings using bucket sort. Character strings must be sorted by more than one key (first by the first character, then the second character, etc.). Bucket sort requires each key to have a single value that maps it to the appropriate bucket.

[2] Signed integer values have to be adjusted for indexing B. For example, for signed 8-bit integer keys between -128 and 127, one simply adds 128 to the keys before indexing the array; this maps the key into the range of array offsets [0...255].

Variations of Bucket Sort

There are several variations on the bucket sort which allow some of the above limitations to be overcome, for example, we might assign keys to buckets using only the k most significant bits of the key. This would assign non-equal keys to buckets, then each bucket could be sorted with another sorting algorithm. This approach can sufficiently reduce the number of required buckets so that bucket sorting a sequence of 32-bit integers is feasible. These variations are beyond the scope of the course, though the interested reader could learn more at the Wikipedia entry for bucket sort as a starting point.

20.1.2 Radix Sort

Radix sort overcomes the space limitations of bucket sort. Radix sort divides each sort key into its individual digits (like we did for trie trees). Strings are divided into characters, integers are divided into their digits from 0 to 9, binary values are divided up into their bits of 0's and 1's. The number of different possible values for a digit of a key is called the *radix*.

In one sense, radix sort is similar to quick sort except that we partition the keys into multiple sets instead of just two, one set for each possible value of a digit. However, instead of comparing keys to perform the partitioning, we index into an array, like in bucket sort.

There are two flavours of radix sort, depending on whether we start partitioning at the most significant digits and work our way to the least significant digits or vice versa. If we start by partitioning on the most significant digits, then we have *MSD radix sort* (most-significant-digit radix sort). Conversely, if we start by partitioning on the least significant digits then we have least-significant-digit radix sort or *LSD radix sort*.[3] Remember: the left-most digit of a key is the most significant digit, and the right-most digit is the least significant digit.

MSD Radix Sort

Suppose we are sorting 32-bit unsigned integer keys. The radix, R, is 10, because we have ten possible values for each digit of our keys. MSD radix sort begins by creating an array $list$ of R empty lists. We append each key to the list $list[k]$ where k is the value of the first digit of the key. Then for each $list[k]$ we recursively sort it based on the next-most-significant digit (or, if $list[k]$ is quite short, we can sort it more quickly using insertion sort because insertion sort has less overhead and does not need to create new empty lists). Once each list has been recursively sorted, we can concatenate each $list[k]$ in order from $k = 0$ to 9 to recover the sorted sequence. This works because each list has already been sorted by its least significant digits, so ordering them by the next most significant digit produces the correct ordering. The MSD radix sort algorithm follows. When considering this algorithm, note that MSD radix sort always sorts keys in *lexicographic order*. This is true even of integer keys, which means that **integer keys are not necessarily sorted in increasing numeric order!** While 4 will appear in the output before 42, the number 320 will appear before **both** 4 and 42 because the first digit of 320 is smaller than the first digit of 4 and 42. This is for the same reason that all words starting with 'a' appear before all words starting with 'b' in the dictionary, regardless of their length. That's lexicographic order! For this reason, MSD radix sort is ideal for sorting strings. If a sequence of integer keys are all of the same length, however, the lexicographic order coincides with the numeric order, and the integer keys are sorted in the order we would normally expect.

[3] Not as hallucinogenic as it sounds.

```
Algorithm MsdRadixSort(keys, R)
keys - keys to be sorted
R - the radix

// Initiate recursive procedure
sortByDigit(keys, R, 0)


Algorithm sortByDigit(keys, R, i)
keys - keys to be sorted
R - the radix
i - digit on which to partition -- i = 0 is the right-most digit

    // all the keys have the same digit sequence for digits 0, 1, ... , i-1,
    // so start by sorting using the possible values for digit i */
    for k = 0 to R-1
        list[k] = new list  // Make a new list for each digit

    for each key
        if the i-th digit of the key has value k add the key to list k

    for k = 0 to R-1
        if there is another digit to consider
            if list[k] is small
                use an insertion sort to sort the items in list[k]
            else
                sortByDigit(list[k], i+1)

    keys = new list // empty the input list

    // Now rebuilt rebuild it so it is sorted in order by
    // the R-i least significant digits.
    For k = 0 to R-1
        keys = keys append list[k]
        // list[k] is already sorted on its R-i-1 least significant digits.
```

Note that, as written, the algorithm given MSD radix sort algorithm inherently assumes that every key has the same number of digits. It is fairly easy to take into account the case where this is not true by having an `list` that holds the keys that don't have any more digits, and in the last loop, appending that list to `keys` before any of the recursively sorted lists, thus causing, say, 1 to appear in the output sequence before 11.

LSD Radix Sort

Again suppose 32-bit unsigned integer keys. LSD radix sort again uses R temporary lists, one for each possible digit. The algorithm looks a lot like a series of bucket sorts on the individual digits, starting with the least significant digit and moving, one at a time, to the most significant digit. To begin, each key, taken from the input array from left to right, is placed on $list[k]$ where k is the key's right-most digit. Then the R lists are concatenated so that the keys are sorted by the least significant digit and are stored back in the input array. This process is repeated for successively more significant digits, ending with the most significant digit. Since each sort on each digit is stable, the partial orders established by the sorts on less significant digits are not destroyed by the subsequent sorts. After the d-th least significant digit has been processed, the keys will be sorted by their d least significant

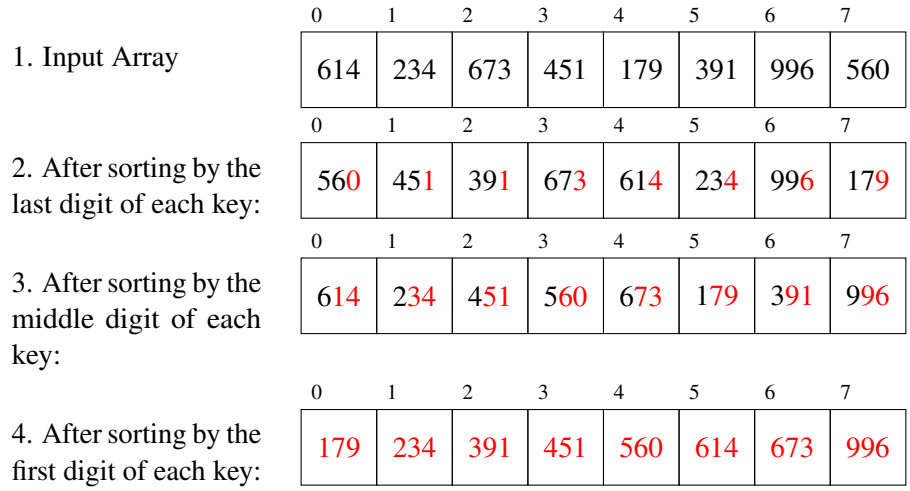
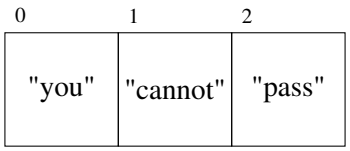


Figure 20.1: Three passes of LSD radix sort for an array of 3-digit integer keys. The red digits show that after the d-th pass, the keys are sorted by their d least significant digits.

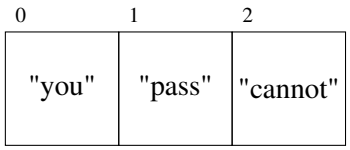
digits. Figure 20.1 illustrates radix sort for a set of 3-digit integer keys.

```
Algorithm LsdRadixSort(keys)
keys - array of keys to be sorted

For each digit d from least significant to most significant
    /* keys are already sorted by digits d+1, d+2, ...
     * so now use a stable sort to sort by digit d */
    for k = 0 to R-1  // for each possible value of digit d
        list[k] = new list

    for each key in order from first to last
        if the d-th digit of the key has value k
            add the key to the end of list[k]
    keys = new list  // Empty the list 'keys'

    for k = 0 to R-1
        keys = keys append list[k]
```

LSD radix sort always puts integer keys into the correct, numeric order. However, LSD radix sort does not sort strings in lexicographic order. LSD radix sort will always place a shorter key before a longer one (which is why numbers come out in the right order as opposed to MSD radix sort!). Thus, the array:



would be sorted by LSD radix sort as:



which is not lexicographic order! LSD radix sort is, therefore, not normally a good candidate for sorting strings.

Stability

MSD radix sort is not stable, but LSD radix sort is.

Time Complexity

Deriving the time complexity of MSD radix sort is a bit tricky since it requires the analysis of a recursion tree like quick sort and merge sort. We won't go into the details, but a sketch of the argument follows thusly: At each level of the recursion tree, there is an $O(R)$ cost to initialize the lists, an $O(n)$ cost to place each element in a temporary list (not per recursive call but per **level** of the recursion tree), and an $O(n)$ cost to concatenate the recursively sorted temporary lists. This amounts to $O(R)+O(n)+O(n)$ or equivalently $O(n+R)$ time cost per level of the recursion tree. Since each recursive call considers the next digit, the height of the recursion tree is no more than the number of digits in the longest key. Thus the total time complexity is $O((n+R)\cdot \text{max. length of key})$. Since the length of keys are usually small (32-bit integers cannot represent keys with more than 10 digits) the length of the key is effectively a constant, and the MSD radix sort effectively requires $O(n+R)$ time.

The time complexity of LSD radix sort is easier to derive, and can be done using the methods we are already familiar with. The algorithm consists of an outer loop and three inner loops in sequence. The first inner loop initializes a list for each of the R digits; this costs $O(R)$. The second inner loop places each key on one of these lists. There are n keys, so this costs $O(n)$. The third inner loop concatenates the lists formed by the second loop which involves copying each key once back to the original array; this costs $O(n)$. Thus, each iteration of the outer loop of LSD radix sort costs $O(R)+O(n)+O(n)$ or just $O(n+R)$. The outer loop happens d times where d is the length of the longest key. Thus, LSD radix sort is $O((n+R)\cdot \text{max. length of key})$. Again, since key lengths are rarely very large, the maximum length of the key is effectively a constant, and LSD radix sort is $O(n+R)$.

Since R is independent of n, for a specific type of data, say, integers, R is fixed, so the time it takes to sort n integers depends only on n, both LSD and MSD radix sorts are effectively $O(n)$ sorts.

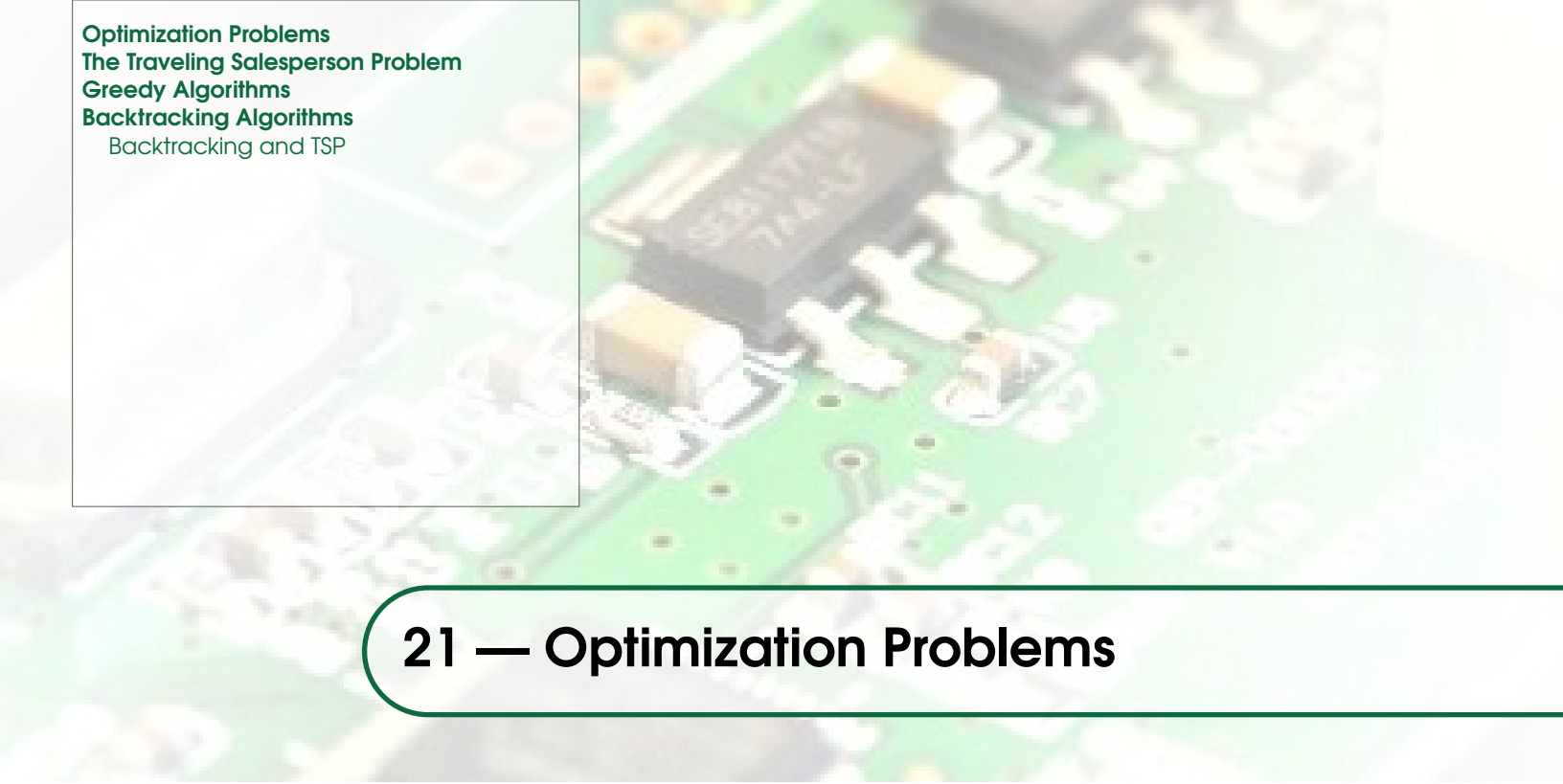
Optional Further Reading

If you're interested in insanely fast radix sorts using GPU-based implementations (i.e. sorting using a graphics card), you might find these results interesting. This page talks about a GPU implementation of radix sort in CUDA that can sort keys on the order of hundreds of millions of keys per second. That means you can sort one million keys in less than 10ms.[4] The exact time cost depends on the GPU hardware, the size of the keys (affecting how many can fit in the GPU's limited memory) and the size of the elements associated with each key.

[4] Try doing **that** on the CPU!

20.2 Other Sorting Algorithms

Although we have covered the common, most well known sorting algorithms, there are plenty of sorting algorithms that we will not cover. However, interested parties will find Wikipedia entry on Sorting to be a useful starting point for further reading if desired.



21 — Optimization Problems

Learning Objectives

After studying this chapter a student should be able to:

- define the Travelling Salesperson Problem (TSP);
- appreciate the difficulty of the TSP;
- explain the general concepts of *greedy algorithms* and *backtracking*;
- explain the advantages and drawbacks of both greedy algorithms and backtracking algorithms; and
- explain how these general algorithmic approaches can be used to develop solutions to the TSP.

21.1 Optimization Problems

An *optimization problem* in computer science is one in which there are many possible or *feasible* solutions, but we are seeking the optimal or best solution. An example of an optimization problem is finding the *minimum spanning tree* of a graph. Recall that a spanning tree of a weighted graph is just a subgraph that is a tree, and contains every vertex of the original graph. The **minimum** spanning tree is the spanning tree from among all of the possible spanning trees that has the smallest sum of edge weights. Another example of an optimization problem is the single-source shortest path problem (solved by Dijkstra's algorithm) which is actually several optimization problems at once: for all $v \in V$, find the shortest path from a start vertex u to v.

Optimization problems are usually much harder than the corresponding "existence" problems. Finding **a** spanning tree is much easier than finding **the minimal** spanning tree. Any depth-first search tree of a connected graph is also a spanning tree, so it is easy to find one. But to find the smallest one, you have to either check them all, which is very expensive (imagine having to compute the sum of the edge weights in every possible depth-first search tree from every possible start node), or find a more clever way to find the optimum solution (e.g. Prim's algorithm or Kruskal's algorithm).

Some optimization problems are harder than others. Minimum spanning tree and single-source shortest paths are examples of very easy optimization problems; they have polynomial-time solutions. Interesting optimization problems are much more likely to be in the class of NP problems, and not

in the class of P problems.[1] Problems in NP but not P do not have polynomial time solutions on silicon computers. In this chapter, we will look at two algorithmic paradigms for designing solutions to otherwise intractable optimization problems: *greedy algorithms* and *backtracking*. Perhaps the most famous optimization problem in NP is the *travelling salesperson problem*, so we will use that problem to illustrate these concepts.

21.2 The Traveling Salesperson Problem

The travelling salesperson problem (or TSP) can be stated thusly: Given a list of cities, and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns back to the starting city. I lied a little before — the problem as stated is actually an NP-hard problem, which means that it is **at least** as hard as the hardest problems in NP (it might be in NP, but it might not, it might be even harder!).

The TSP is typically formulated as a problem on a weighted undirected graph $G = (V, E)$. Nodes in V represent the cities, and the weighted edges represent connections between cities. If it is possible to travel directly between city u and city v, then there is an undirected edge $[u, v] \in E$ between them, and the weight of this edge represents the cost of following that edge (which could represent distance, travel costs, etc.). A Hamiltonian Path is a path that visits each vertex in the graph exactly once. A Hamiltonian cycle results from connecting the last node in a Hamiltonian path back to the first node. Thus, when the TSP is formulated as a graph problem, the problem is reduced to: find the Hamiltonian cycle in the graph with the smallest sum of edge costs. Equivalently we could search for the cheapest Hamiltonian path because every Hamiltonian cycle is just a hamiltonian path with the last edge removed.

How hard is this problem? Let's suppose the worst case, that direct travel is possible between every pair of cities. How many Hamiltonian paths are there in such a graph? Well, we can start from any node we please. If there are n nodes in the graph, we can choose any of those n nodes as the start node. Now, how many choices are there for the second node in the path? We cannot visit a node more than once, so that means we could choose any of the $n-1$ remaining nodes as the second node. Thus, there are $n \cdot (n-1)$ choices for the first two nodes. Continuing in this fashion, there are $n \cdot (n-1) \cdot (n-2)$ choices for the first three nodes, $n \cdot (n-1) \cdot (n-2) \cdot (n-3)$ choices for the first four nodes, and so on, until we are left with one possible choice for the last node. Thus, the total number of possible Hamiltonian paths is:

$$n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot \cdots \cdot 2 \cdot 1 = n!$$

In other words, in the worst case, to solve the TSP we need to find the cheapest Hamiltonian path from among $n!$ possible or *feasible* solutions. Yeah... good luck with that. If we have to enumerate every possible Hamiltonian path and check if it has the minimum cost, the problem gets impractical already for a graph with only 13 nodes, for which we would have to check about 6.7 billion paths.

In the next sections we will look at greedy algorithms and backtracking algorithms. We will see that greedy algorithms can sometimes give us *approximate* solutions to NP problems in polynomial time. That is, algorithms that can find really good solutions, but with no guarantee that the solution is, in fact, optimal.

[1] Problems in P have a polynomial time solution on silicon computers. NP or nondeterministic-polynomial-time problems are those than can be solved by **nondeterministic computers** in polynomial time. Silicon computers are **deterministic**. Problems that are in NP but not P cannot be solved by deterministic computers in less than exponential time.

21.3 Greedy Algorithms

Greedy algorithms can be characterized thusly:

- Start with a partial solution.
- Repeatedly extend the partial solution until a complete solution is obtained.

Greedy algorithms begin by choosing a partial solution. The initial partial solution is usually trivial. In the case of the TSP, we might begin with an empty path. A greedy algorithm then repeatedly extends the partial solution to a more complete solution, subject to feasibility constraints, until a complete solution is obtained. In the case of the TSP, this would mean somehow choosing the next node in the path such that the solution is likely to be a very low-cost path, with the feasibility constraint that we cannot choose a node that is already in the path.

In general, greedy algorithms all look like this:

```
Start with a (usually trivial) partial solution

While the solution is not complete
    Extend the partial solution to a more complete solution in a way that
    seems like it should lead to a good solution while maintaining
    feasibility.
```

For a given problem, there can be many different greedy algorithms depending on:

- the form that the partial solution takes;
- the choice of the initial partial solution; and
- the method for extending the partial solution to a more complete one.

To illustrate how these things might differ, let's consider the optimization problem:

Given a weighted graph $G = (V, E)$, and two nodes $s, t \in V$, find the shortest path from s to t.

One possible greedy algorithm solution is:

Shortest path from s to t - Greedy Algorithm #1
Partial Solution Form: a partial shortest-paths tree from s.
Initial partial solution: the start vertex, s.
Extension: Select the edge that meets the following criteria:

- exactly one endpoint of the edge is already in the shortest-paths tree; and
- the path from s to the selected edge's endpoint that is not in the tree is as short as any other path from s to a non-tree vertex (this is the greedy part — select the extension that is *locally* the cheapest!).

When the shortest-paths tree is complete, the path from s to t in the tree is the solution.

If we think about this solution carefully, we'll realize that it is actually... Dijkstra's algorithm! It turns out that Dijkstra's algorithm is a greedy algorithm which just **happens** to also guarantee the optimal solution! Remember, not all greedy algorithms guarantee the optimal solution to an optimization problem. To see that this is true, let's look at a different greedy algorithm for the same problem of finding the shortest path from s to t.

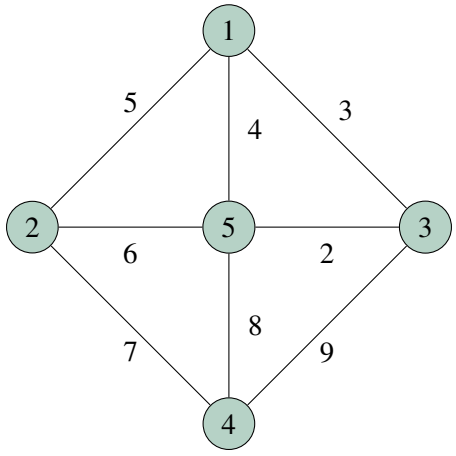
Shortest path from s to t - Greedy Algorithm #2
Partial Solution Form: a partial shortest-paths tree from s.
Initial partial solution: the start vertex, s.
Extension: Select the edge that meets the following criteria:

- exactly one endpoint of the edge is already in the shortest-paths tree; and
- the edge has the smallest weight (again, this is the greedy part).

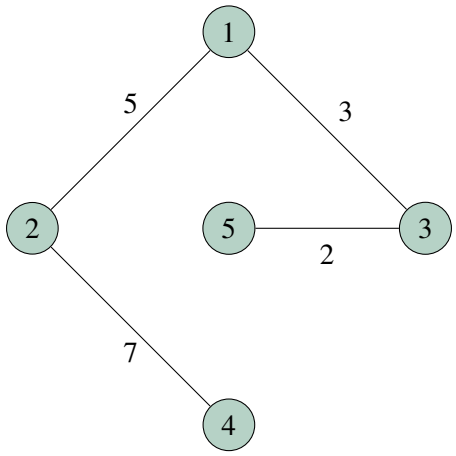
When the shortest-paths tree is complete, the path from s to t in the tree is the solution.

Note that the only difference is the change in the criteria: we still extend the partial solution by adding an edge that has exactly one endpoint already in the tree, but from among the possible candidates meeting that criteria we instead now choose the edge with the smallest weight (instead of the edge that creates the shortest **path** from s). This version of our greedy algorithm should also be familiar. It is, in fact, exactly Prim's algorithm for minimum spanning tree!

If we apply our second greedy algorithm to the following graph, we will see that it finds a good path from node 1 to node 5, but not the optimal path!



Initially node 1 is in the shortest paths tree. Then we add the cheapest edge that is incident to 1. That is edge $[1,3]$. Now nodes 1 and 3 are in the tree, and we need to add the cheapest edge that is incident to nodes 1 or 3. This happens to be edge [3,5]. Now nodes 1, 3, and 5 are in the tree, and the cheapest edge incident on exactly one of 1, 3, or 5, is $[1,2]$. Now every node is in the tree except for node 4, so we add the cheapest edge to 4, which is $[2,4]$. The resulting shortest paths tree is:



This tells us that the shortest path from 1 to 5 is the path 1-3-5 with cost 5. This path is pretty short compared to some of the other possibilities, but it is clearly not the **optimal** path! The optimal path is to go directly from node 1 to node 5 with a cost of 4. What went wrong? The greedy nature of the algorithm made a locally optimal decision. Choosing to start with edge $[1,3]$ instead of edge $[1,4]$ (because edge $[1,3]$ was cheaper) forced edge $[3,5]$ to be added next because it had a cost of only 2, which caused node 5 to become part of the shortest-paths tree. From this point forward in the algorithm, edge $[1,5]$ cannot ever be considered because both of its endpoints are now in the tree. This is what greedy algorithms do! They make quick, myopic decisions based on what looks optimal at the time, but might end up not being globally optimal because they do not consider the larger context of such decisions. Essentially, we are trading accuracy for speed. We will still get a "pretty good" solution, but with no guarantee that it is optimal.

At this point, you might be wondering: why would we ever use Greedy Algorithm #2 for shortest paths? The answer is: well, we wouldn't. We'd just use Dijkstra's algorithm because it guarantees the optimum. The big advantage of greedy algorithms becomes apparent when we look at harder problems like TSP. A greedy algorithm can get us a good solution to TSP in polynomial time, but without a guarantee of optimality. There are no known polynomial-time greedy algorithms that can guarantee an optimal solution to the TSP.[2] There are, however, polynomial-time greedy algorithms for TSP that can get us a good approximation of the optimum solution, and may even sometimes give us the optimum solution, but without any guarantee that it **always** will. We will look at two such algorithms in class.

21.4 Backtracking Algorithms

Backtrack programming, or just *backtracking*, is another algorithmic approach to solving optimization problems. Unlike greedy algorithms, backtracking algorithms **do** guarantee that the optimal solution is found. If that's true, how do they differ from the naive algorithm of "try every possible solution, find the best one"? Backtracking does perform an exhaustive search of all possible solutions in a systematic way, but its main feature is to eliminate partial solutions that can never lead to an optimal solution. This means that while the optimal solution is always found, we will not have to look at **every** possible solution in order to find it. For example, we might discover that a partial path s-A-B-C from s to t in a graph from a start node is already more costly than a previously found path

[2] If there were, it would imply that P=NP!!!

from s to t, which means we can immediately stop considering any longer path that begins with s-A-B-C, potentially eliminating a lot of possible solutions at once. Backtracking can also eliminate partial solutions that are infeasible (don't meet some constraints) which means that again, we can eliminate all possible solutions that could be built from the infeasible partial solution.

Even with these improvements, backtracking algorithms usually require exponential time, that is, $O(a^n)$ for some real constant a. The general approach to all backtracking algorithms is expressed in the following pseudocode:

```
driver method:
    best solution = nil
    T = initial partial solution
    testAllExtensions(T)
    output the best solution

method testAllExtensions (T):
    if T is a complete solution
        if T is better than the current best solution
            save T as the best solution
    else for each possible extension of T
        apply the extension to T
        if feasible(T) and T has potential to be better than current best
            testAllExtensions (T)
        remove the extension from T
```

Note how the recursive call in `testAllExtensions` is only invoked if the current extension to the partial solution is both feasible, and has potential to be better than the best solution found so far. If the condition is false, the recursive call is not made, which immediately eliminates all solutions that are extensions of the current partial solution from consideration because they don't have to be checked. The search then continues using a different extension of `T`.

21.4.1 Backtracking and TSP

We can apply backtracking to the TSP problem to develop an exponential time solution that is guaranteed to find the optimum solution. We could formulate the backtracking solution thusly:

Partial Solution Form: a walk from start vertex s to some vertex x
Initial partial solution: the walk containing only s
Complete solution: a partial solution that contains n vertices is a complete solution.
Feasible solution: a partial solution that is a path (no repeated vertices) is a feasible solution.
Potential to be better than current best solution: if the cost of the partial solution + estimate of cost for remaining edges < cost of best solution so far

A big question in the above approach is how to estimate the cost of the remaining edges. We will discuss a few possibilities here:

Estimate #1: 0. We could only reject a path if it is **already** more expensive than the best solution.

Estimate #2: number of remaining edges $\times$ min. cost of any edge in the graph. This estimate puts a lower bound on the cost of the remainder of the path using the global minimum edge cost. This global minimum edge cost can be pre-computed before backtracking begins. This estimate would potentially allow us to reject partial solutions earlier than Estimate #1.

Estimate #3: for each vertex v not in the partial solution, find the edge incident to v with the least cost, find the sum of all such edges. This estimate provides a better estimate (tighter lower bound) of the cost of the remaining edges, potentially allowing even earlier rejection of partial solutions. However, it will take more time to compute Estimate #3 because it has to be recomputed for every partial solution.

Note that the earlier we can reject a partial solution, the more possible solutions we do not have to check! For example, suppose our input graph has five nodes. If we reject a partial solution of 4 nodes, we aren't saving much because there was only one possible solution with those four nodes as the first four nodes. However, if we can reject a partial solution with 2 nodes, there are $3 \times 2 \times 1 = 6$ solutions we don't have to check. In general, if we reject a partial solution with k nodes, then we eliminate the need to check $(n-k)!$ complete solutions.

22 — Case Study: Filesystem Data Structures

22.1 Filesystems

The purpose of a filesystem is to provide a mapping from a filename to the physical locations in which that file exists on a storage device. In most cases, filesystems map files to logical blocks, which are then mapped to physical blocks of storage on the storage device.

Keep in mind that we will describe the main features of filesystems with emphasis on the underlying data structures. We will gloss over a lot of the fine-grained technical details and the fact that most of the filesystems we will discuss have several variants.

22.2 FAT Filesystems

The FAT (File Allocation Table) filesystem has its origins in the 1970s as a filesystem for floppy disks. Its main feature is a File Allocation Table which assigns logical *clusters* to physical *sectors* on a disk. A *cluster* is a fixed-size grouping of physical disk sectors. The FAT is statically allocated at the time the disk is formatted.

The disk layout in a FAT filesystem is as follows. The first physical disk sector (sector 0) is a special sector called the *boot sector*. The boot sector contains vital information about the configuration of the FAT filesystem. The sectors immediately following the boot sector contain the FAT. The length of the FAT is stored at a specific fixed byte-offset within the boot sector. The sector(s) following the FAT contain the root directory. The number of root directory entries is fixed, and is stored at a specific byte-offset of the boot sector. All remaining sectors on the disk after the root directory are the *data area* of the disk. These sectors grouped into clusters which are assigned to files via the FAT.

An in-use directory entry contains information about a file, including the file name, attributes, creation date, and the cluster number of the first cluster that belongs to the file. The FAT is essentially an array of integers with one entry for each cluster in the data area, that is, the FAT is indexed by cluster numbers. The integer $FAT[i]$ is one of three things:

- a special value indicating that cluster i is not in use (not assigned to any file);
- the cluster number of the next cluster in the file to which cluster i belongs; or

- a special value called EOF (end of file) indicating that cluster i is the last cluster in the file to which cluster i belongs.

Thus, to find the all of the clusters belonging to a file, the operating system reads the directory entry for the file, and obtains the cluster number of the first cluster, i_1 from the directory entry. Cluster i_1 is the first cluster. To see if the file has additional clusters, the FAT is consulted, specifically $FAT[i_1]$. $FAT[i_1]$ will either contain the cluster number of the next cluster in the file, i_2, or the special value EOF to indicate that there are no additional clusters. If there is a second cluster, i_2, then $FAT[i_2]$ is consulted to see if there are more clusters, and so on in this fashion. Thus, directory entries contain cluster numbers that are the beginnings of chains in the FAT. If a cluster j doesn't belong to any file, then $FAT[j]$ will not be reachable from any chain beginning from any directory entry in the filesystem. Figure 22.1 shows an example of some entries in the root directory and their entries in the FAT. Notice how files could be contiguous and sequential (e.g. IO.SYS and MSDOS.SYS), or they could be fragmented (like COMMAND.COM). The clusters could even be out of sequential order (like JANEWAY.TXT). As files get deleted, leaving holes in the FAT, and new files get allocated across those holes, the FAT filesystem tends to get more and more fragmented over time. This is why "defrag" or *defragmentation* software exists and is popular for FAT filesystems. Fragmented files increase access time because the disk read head has to move further to read each successive cluster belonging to a file and/or the system may have to wait longer for the required cluster to pass under the disk's read head.

Subdirectories are just special files that are assigned clusters via the FAT just like any other file, except that the clusters contain more directory entries.

There have been many variants of FAT over the years to support larger and larger filesystems. Initially, the FAT was indexed with only 8-bit cluster numbers. This was fine for floppy disks; this allowed almost 2^8 clusters, which, assuming, say, 512 bytes per cluster, meant a theoretical capacity of $2^{17} = 128$KiB, a typical size for the old 5.25-inch floppy disks. Later, higher capacity 3.5-inch floppies used a 16-bit FAT, again with small clusters of around 256 or 512 bytes which theoretically allowed for capacities up to a few megabytes. As hard disks became mainstream and grew larger and larger into the hundreds of megabytes, the 16-bit FAT became problematic — cluster sizes increased to be as large as 64KiB. While this allowed for a theoretical maximum capacity of 2^{16} clusters $\times 2^{16}$ bytes per cluster $= 2^{32}$ bytes $= 4$GiB, it still wasn't enough, and cluster sizes couldn't go much bigger. The problem with large clusters is that clusters are the smallest unit of space that can be allocated to a file, thus a file containing only 1 byte of data would still take up 64KiB of data on your hard drive! With the arrival of Windows 95 Service Release 2, the FAT32 filesystem was introduced. Cluster numbers became 32-bits (of which 28-bits were actually used to hold the cluster number) allowing for disks of up to 2 to 16 Terabytes (depending on the physical sector and cluster size). Even on FAT32, the maximum size of a **single file** is 4 GiB, because the file size in directory entries is limited to a 32-bit number, though there are many extensions to FAT32 which get around this, and other limitations.

For further reading on FAT filesystems, you might start at the Wikipedia page, which was used as a source for some of the information in this section.

22.3 UNIX-like Filesystems

Most UNIX-like filesystems are similar in structure, though vary considerably in the fine details. As a somewhat canonical example, we will look at the *Extended Filesystem 2* or *ext2*. ext2 was designed for, and is still supported by, the Linux operating system.

Name	Ext	Size (bytes)	1st Cluster
IO	SYS	5042	0
MSDOS	SYS	3255	5
KIRK	TXT	50	29
PICARD	TXT	1701	42
JANEWAY	TXT	8200	90
COMMAND	COM	17177	9

	0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	EOF	6	7	8	EOF	10
10	13	12	14	15	17	16	19	18	EOF	20
20	21	22	23	24	25	26	27	28	30	EOF
30	31	EOF								
40			87							
50					55	11				
60										
70										
80								EOF		
90	91	54								

Figure 22.1: Schematic of root directory (simplified) and File Allocation Table assuming a cluster size of 1KiB (1024 bytes). Colours are added to make it easier to see which clusters belong to which files.

In ext2, the disk space is organized into logical *blocks*, which are, at some level that is irrelevant to our discussion, mapped to physical disk locations. Again, glossing over many details, a UNIX directory contains entries for each file in the directory, associating each file with an *inode*. An inode is a special disk block that contains information on which disk blocks belong to the file. Like FAT, directories are stored just like files; they have an inode that says which blocks belong to the directory and contain directory entries.

The inode for a file contains some basic information about the file, such as its size, its file attributes, permissions, and ownership, as well as information on which disk blocks belong to the file. An inode in ext2 contains twelve *direct block* pointers. These are the block numbers for up to the first twelve *data blocks* belonging to the file and containing its data. Then there is a block number that points to an *indirect block*. This is a disk block that contains nothing but pointers to the next disk blocks in the file. Suppose the block size for a disk is 1024 bytes, and block numbers are 32 bits. The first twelve direct block pointers are enough for any files up to 12 KiB in size. The indirect block can hold up to 1024/4 = 256 block pointers, so the indirect block can point to the next 256 blocks in the file. This is sufficient for files up to 256 + 12 KiB in size. For larger files, the inode contains a pointer to a *double indirect block*. The double indirect block contains (under our block size assumptions) 256 pointers to 256 more indirect blocks, each of which contain 256 pointers directly to data blocks belonging to the file. That's enough for an **additional** $256 \times 256 \times 1\text{KiB}$ = 65536 KiB or 64 Megabytes of capacity. Finally, if that isn't enough, the inode contains a *triple indirect block* pointer which refers to a block that contains (again, assuming 1024-byte blocks) 256 pointers to double-indirect blocks, each of which contains 256 pointers to indirect blocks, each of which contains a pointer to a data block, for an **additional** $256 \times 256 \times 256 \times 1\text{KiB} = 2^{24} \times 1\text{KiB}$ = 16 Gigabytes of capacity. Figure 22.2 shows the inode and block pointer structure for a single file.

The upshot of this is that blocks for very small files can be accessed quickly with minimal disk accesses. For larger files, blocks beyond the first few require more intermediate blocks, and therefore more disk accesses to get to. This is a good thing because the vast majority of files on most filesystems are quite small.

The number of inodes in many unix-like filesystems is fixed at format-time, imposing a maximum number of files and directories that can exist in the filesystem.

Let's revisit directory files for a moment. Directories are flat lists of files. When an application requests access to a file in a certain directory, the ext2 filesystem performs a linear search (yes, you read that right!) on the directory for the file name. This is fine for relatively small directories, but it is horrendous for directories containing a large number of files. Fortunately, later versions of the extended filesystem (e.g. ext3 and ext4) can store directory entries in a tree structure called an HTree to improve search times. We leave the investigation of HTrees to the reader, but we can remark that they are similar in some respects to B-Trees in that they are shallow trees with a high branching factor.

There are so many flavours of UNIX-like filesystems, all of which use something similar to the inode structure we've shown here. One interesting variant is XFS, which is a 64-bit filesystem that can theoretically support filesystems with capacity up to 8 exabytes.

22.4 Apple HFS

In this section we'll look at Apple's Hierarchical File System (HFS). In HFS, logical sectors (usually 512 bytes) are grouped together into allocation blocks.

The main parts of HFS (for the purposes of our discussion) are:

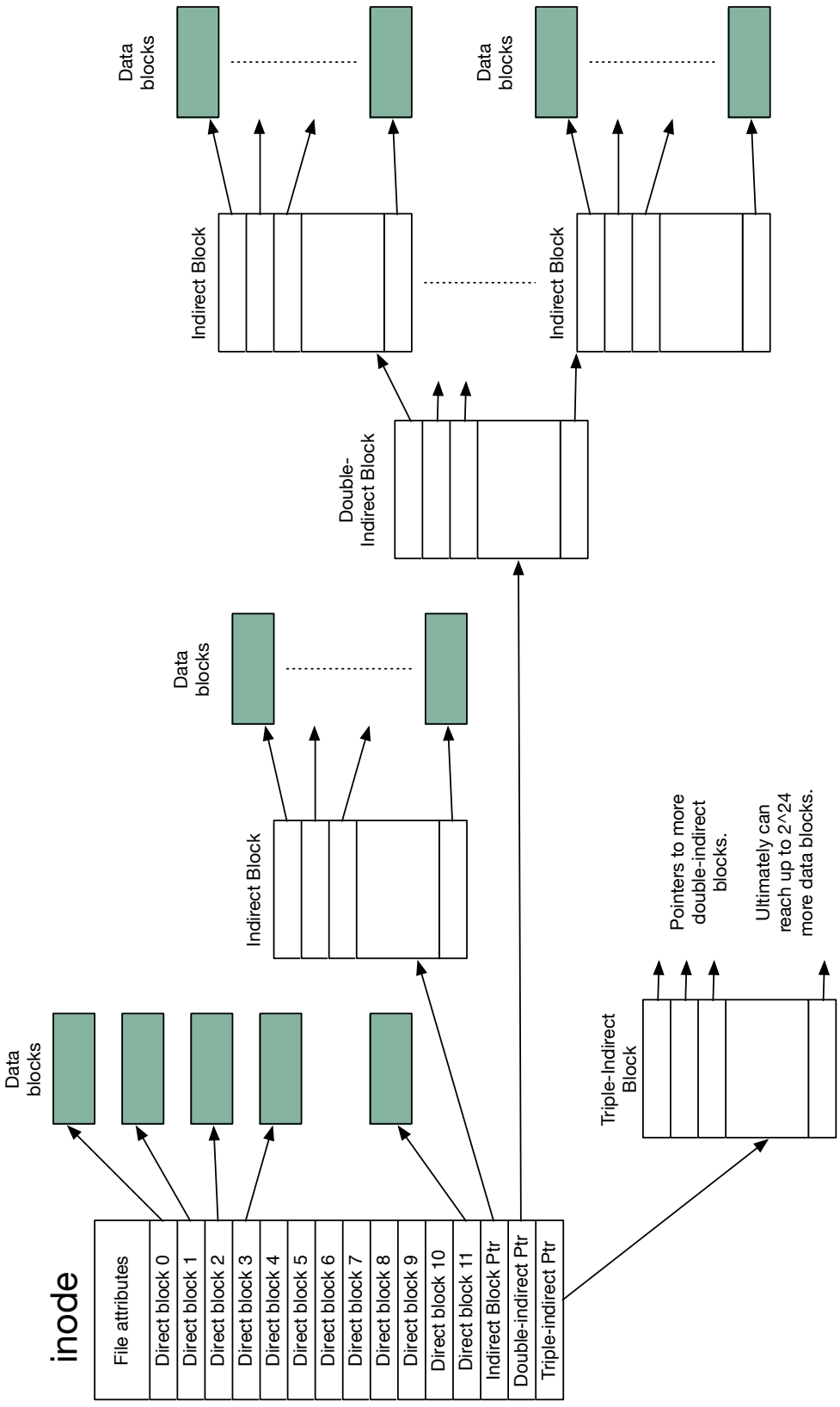


Figure 22.2: An inode and its block pointer structure in ext2.

- a volume bitmap that indicates which allocation blocks are allocated to a file, and which aren't. The size of the volume bitmap is determined by the number of allocation blocks on the storage device.
- a *catalog file* which contains a record for every file and directory in the filesystem storing metadata, and information on allocation blocks assigned to the file.
- An *extent overflow file* which contains information about additional allocation blocks that are assigned to files (more on this shortly).

The Catalog File

The *catalog file* is a B-tree. This B-tree stores a record for every single file and directory in the filesystem. The records are keyed on a unique Catalog Node ID (CNID). A file record in the B-tree contains various attributes and metadata about a file, including its CNID, and its size. Additionally, the file record stores the first three *extents* of the file. An *extent* is a contiguous sequence of allocation blocks on the storage device. Unlike FAT or ext2, HFS doesn't explicitly store every allocation block that belongs to a file, instead it stores extents, which are essentially ranges of contiguous allocation blocks. A file record in the Catalog File can contain up to three extents. But what if a file occupies more than three different contiguous sequences of allocation blocks on the storage device? We'll answer that in the next section.

Information about directories are stored in directory records in the catalog file. These are very similar to file records, but the metadata differs.

The Extent Overflow File

The *extent overflow file* is another B-tree which is used when a file occupies more than three extents on the storage device. When the three extents in the file record in the catalog file are exhausted, additional extents for the file are stored here in records keyed by CNID.

Extensions

The current version of Apple's filesystem is HFS+ which offers several improvements over the original HFS.

In HFS+ the extent overflow file can also store extents of known bad allocation blocks, ensuring that they are never allocated to a file.

HFS+ changes the name of the volume bitmap to the *allocation file*. Unlike the volume bitmap in HFS, in HFS+ the allocation file is stored as a regular file, and can therefore change size, does not have to occupy a special reserved area at the beginning of the storage device, and does not even need to be stored in a single contiguous extent of storage. This greatly facilitates the ability to resize HFS+ volumes without destroying its contents.

HFS+ also adds a new B-tree file called the Attributes file. This allows for extended metadata to be attached to a file.

HFS+ uses 32-bit numbers to address allocation blocks rather than the 16-bit numbers used by HFS, allowing filesystems with greater capacity. HFS+ can address 2^{32} allocation blocks and the theoretical maximum capacity is determined by the number of logical sectors in each allocation block.

22.5 NTFS

New Technology File System, or NTFS, is the default filesystem for modern Windows operating systems. It is very complicated and poorly documented, so we only skim over the very basics of NTFS, even more so than we glossed over details in other sections of this chapter.

In NTFS, at format time a reserved area of the disk (12.5% of its capacity, by default) is set aside to hold entries in the *Master File Table* (MFT). This is done to prevent fragmentation of the MFT.

The MFT is a relational database. Think of this as a giant table, in which rows are file records and columns are file attributes. Relational databases can be searched very efficiently. k-D trees are one example of a way of indexing a relational database. Relational databases look like tables to the user, but are stored much differently for speed of searching.

Every allocated disk cluster in NTFS belongs to a file, even filesystem metadata. Every file and folder has at least one MFT record. The MFT record for a file records its *attributes*. Everything that is not metadata is a file attribute, even the contents of a file itself (the file data). Attributes of a file that are stored in the file's MFT record are *resident attributes*. Attributes for which there is not enough space in the MFT record are allocated clusters on the disk and the MFT contains references to those external clusters. Each MFT record will also have a unique identifier on which the entire relational database is keyed.

MFT records are 1KiB in size. Small files ($<$ 900 bytes or so) can be contained entirely within the MFT. The file data itself is an attribute that is stored right in the MFT record, in addition to such things as standard information (access mode, time stamp, etc.), and the file name. Each specific piece of data is a file attribute. Larger files for which the data cannot fit in the MFT record have references in their MFT record to clusters containing additional file data attributes. Large files might need more attributes than can fit in an MFT record, in which case, the base MFT record has attributes that references other MFT records.

Folders have MFT records that contain index attributes that refer to the MFT records for the files they contain. Small folders are stored entirely within their base MFT record. For larger directories, the MFT record contains pointers to external clusters which organize MFT record pointers of the contents of the folder in a B-tree structure.